const express = require('express');
const app = express();
app.get('/', (req, res) => {
res.send('Hello Node.js!');
});
app.listen(3000, () => {
console.log('Server
running on port 3000');
});
Node.js
COMPLETE
CHEAT SHEET
Modules, npm & Express in 1 Video
Node.js का
सबसे बड़ा
Complete
Guide!
BACKEND
Module
npm
Express
Core Modules
(जरूरी Modules)
npm Commands
(पूरी जानकारी)
Express.js
(Routing, Middleware)
REST API
(बनाना सीखें)
Best Practices
(Pro Tips)
Deployment
(Production Ready)
CORE MODULES
NPM COMMANDS
EXPRESS.JS
DATABASE CONNECTION
DEPLOYMENT
BEST PRACTICES
एक ही वीडियो में सब कुछ सीखें!

Node.js Basics

JavaScript Runtime Built for Scalable Network Applications

JavaScript Runtime
Node.js lets you run JavaScript on the server side.

V8 Engine
Built on Chrome's V8 engine for high performance and speed.

Server Side
Execute JavaScript on the server to build powerful backend apps.

Event Driven
Uses an event-driven architecture to handle multiple requests efficiently.

```
// Hello Node.js
const http = require('http');

http.createServer((req, res) => {
  res.end('Hello Node.js!');
}).listen(3000, () => {
  console.log('Server running on port 3000');
});
```

Non Blocking I/O
Performs I/O operations without blocking the execution.

Cross Platform
Runs on Windows, macOS, and Linux.

Fast & Scalable
Lightweight, fast, and perfect for building scalable network applications.

High Performance
Built for speed and efficiency.

Reliable
Stable and reliable runtime environment.

NPM Ecosystem
Huge ecosystem of packages and modules.

Open Source
Open source and backed by a large community.

Node.js Installation & Setup

From Download to Your First Node.js Project

1 Download Node.js

Go to

https://nodejs.org

and download Node.js for your OS.

2 Choose LTS Version

LTS

Recommended For Most Users

LTS (Long Term Support) version is stable and recommended for production.

3 Verify Installation (node -v)

```
$ node -v
v20.12.2
```

Check Node.js version installed on your system.

4 Verify npm (npm -v)

npm

```
$ npm -v
10.5.0
```

Check npm version (Node Package Manager).

5 Install VS Code

Use VS Code as your code editor for the best development experience.

6 Open Terminal

Use Terminal (or VS Code Integrated Terminal) to run Node.js commands.

7 Create a Project Folder

```
$ mkdir my-node-app
$ cd my-node-app
```

Create a new folder and navigate into it.

8 Initialize package.json

JSON

```
$ npm init -y
```

Creates a package.json file with default settings.

Project Structure

```
my-node-app/
├── { } package.json
├── JS index.js
└── node_modules/
    └── ...
```

What is package.json?

JSON

package.json is the heart of your Node.js project. It contains metadata about your project and its dependencies.

- Stores project info
- Manages dependencies
- Defines scripts
- Helps in sharing your project

Example: package.json

```
{
  "name": "my-node-app",
  "version": "1.0.0",
  "description": "My first Node.js project",
  "main": "index.js",
  "scripts": {
    "start": "node index.js"
  },
  "author": "Your Name",
  "license": "MIT"
}
```

Pro Tip: Use LTS for stability, keep Node.js & npm updated, and always check versions.

`$ node -v`	`$ npm -v`	`$ npm update -g npm`	`$ npm outdated`
Check Node.js version	Check npm version	Update npm	Check outdated packages

You're all set!

Start building amazing things with Node.js 🚀

Node.js Architecture

Event-Driven, Non-Blocking I/O Model

Single Thread
One main thread handles all requests

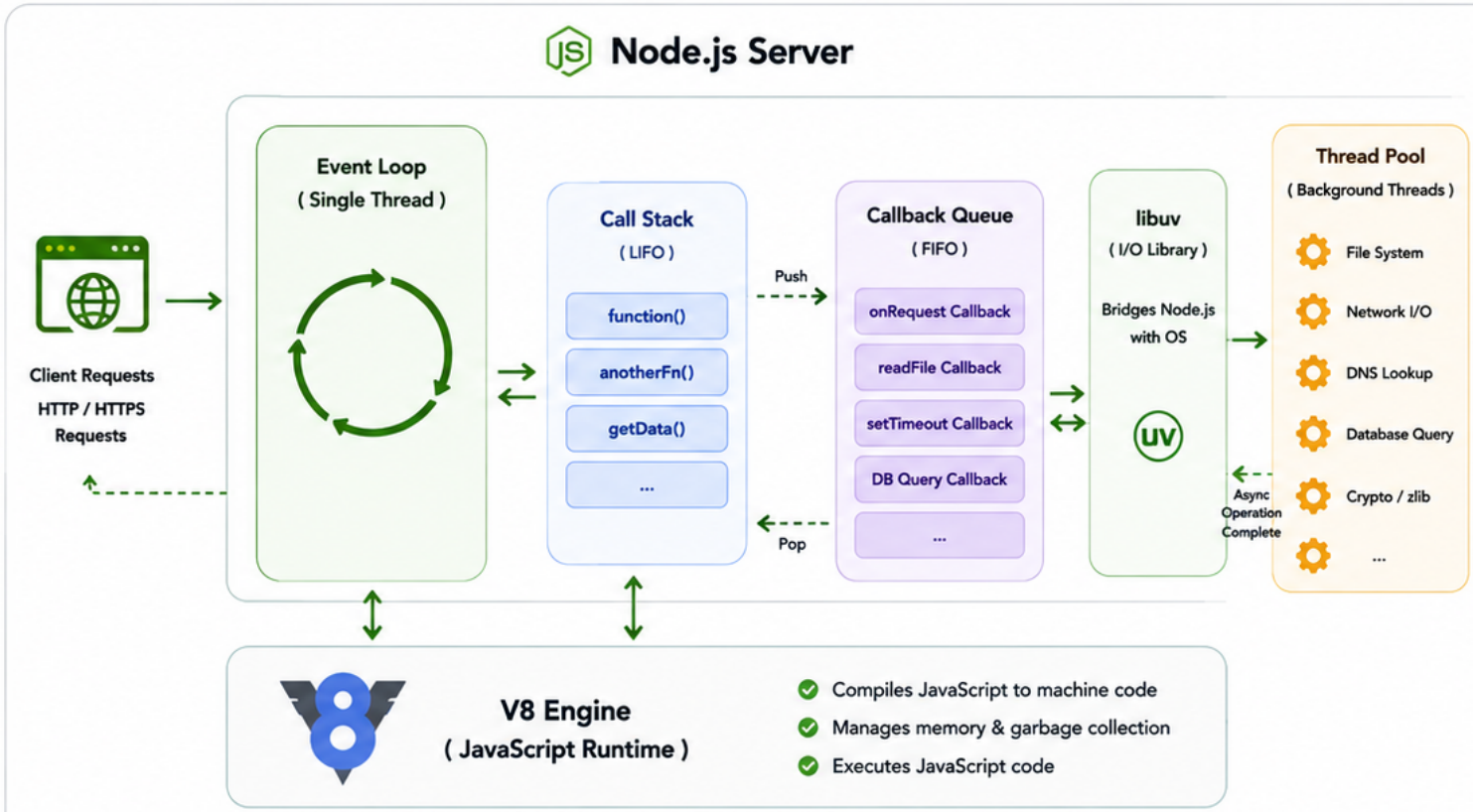
Event Driven
Uses events and callbacks

Non-Blocking I/O
I/O operations don't block the thread

Asynchronous
Enables high concurrency

How It Works

1. **Request comes in**
   A client request arrives at the Node.js server.
2. **Handled by Event Loop**
   Event Loop checks the Call Stack.
3. **If task is async**
   It goes to the system (APIs / DB / File I/O) via **libuv**.
4. **Background Work**
   Thread Pool performs the async operation.
5. **Callback Queue**
   Once done, callback is moved to **Callback Queue**.
6. **Back to Event Loop**
   Event Loop pushes the **callback to Call Stack** for execution.
7. **Response Sent**
   Result is sent back to the client.



Key Components

Event Loop
Continuously checks Call Stack and Callback Queue.

Call Stack
Executes functions. Single threaded (LIFO).

Callback Queue
Holds callbacks ready to be executed (FIFO).

libuv
Handles async I/O operations using OS APIs.

Thread Pool
Performs heavy operations in the background.

V8 Engine
Executes JavaScript code with high performance.

High Performance
Handles many connections with minimal resources.

Scalable
Built for real-time and data-intensive apps.

Lightweight
Minimal overhead and fast execution.

Cross Platform
Runs on Windows, macOS, Linux, and more.

Perfect for
APIs, Microservices, Real-time apps, SPAs.

Node.js Modules

Organize Code. Reuse Logic. Build Better Applications.

Reusability — Write once, use anywhere

Maintainability — Keep code clean and organized

Scalability — Easy to scale large applications

Encapsulation — Hide implementation details

1 Core Modules

Built-in modules that come with Node.js.

fs · path · http · os · events · ...

```
// Example: Using fs (File System) module
const fs = require('fs');
fs.readFile('file.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

2 Custom Modules

Create your own modules and export functionality.

math.js (module) → Export → app.js (use module)

math.js

```
function add(a, b) {
  return a + b;
}

module.exports = add;
```

app.js

```
const add = require('./math');
console.log(add(5, 3));  // 8
```

How Modules Work in Node.js

app.js

```
const math = require('./math');
const msg = require('./message');

console.log(math.add(2, 3));
console.log(msg.greet('Node.js'));
```

require() → math.js

```
function add(a, b) {
  return a + b;
}

module.exports = { add };
```

require() → message.js

```
function greet(name) {
  return `Hello, ${name}`;
}

module.exports = { greet };
```

Node.js Modules follow the CommonJS system by default.
Each file is treated as a module with its own scope.

CommonJS vs ES Modules

Feature	CommonJS (Default)	ES Modules (Modern)
Import	`const x = require('x');`	`import x from 'x';`
Export	module.exports / exports	export / export default
Loading	Runtime (Dynamic)	Compile-time (Static)
File Extension	.js	.mjs or .js (type: module)
This	module.exports	undefined
Use	Default in Node.js (older)	Recommended (modern)

3 require()

Used to import modules.

```
const fs = require('fs');
const myModule = require('./myModule');
```

4 module.exports

Used to export a module as a whole.

```
const users = { getAll() {}, getById() {} };
module.exports = users;
```

5 exports

Shortcut for **module.exports**.

```
exports.greet = function(name) {
  return `Hi, ${name}`;
};
```

6 import (ES Modules)

Used to import in ES Module syntax.

```
import { add } from './math.js';
import pkg from './package.js';
```

7 export (ES Modules)

Used to export in ES Module syntax.

```
export function add(a, b) {
  return a + b;
}
export const PI = 3.14159;
```

8 ES Modules (.mjs / type: module)

- Modern JavaScript module standard
- Static analysis and tree shaking
- Use .mjs extension or set "type": "module" in package.json

package.json

```
{
  "type": "module"
}
```

Best Practices

Keep modules small — Do one thing and do it well.

Use meaningful names — Easy to understand and maintain.

Export only what you need — Keep the module interface clean.

Prefer ES Modules — For new projects and modern codebases.

Check for circular dependencies — Can cause bugs and unexpected behavior.

Node.js Common Core Modules

Built-in modules that come with Node.js

No Installation Needed | Reliable & Fast | Ready to Use

1 fs

File System

Work with the file system. Read, write, update, delete files and directories.

```
// Read a file
const fs = require('fs');
fs.readFile('example.txt', 'utf8',
 (err, data) => {
   if (err) throw err;
   console.log(data);
});
```

2 http

HTTP

Create HTTP servers and make HTTP requests.

```
// Create a server
const http = require('http');
const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello, Node.js!');
});
server.listen(3000, () => console.log('Server running'));
```

3 path

Path

Work with file and directory paths.

```
// Join paths
const path = require('path');
const fullPath = path.join(__dirname,
 'folder', 'file.txt');
console.log(fullPath);
// Get file extension
console.log(path.extname('file.txt'));
```

4 os

Operating System

Get information about the operating system.

```
// OS information
const os = require('os');
console.log(os.platform());
console.log(os.release());
console.log(os.cpus().length + ' CPUs');
console.log(os.totalmem() / 1024 / 1024 +
 ' MB RAM');
```

5 url

URL

Parse and format URLs. Work with URL strings and query parameters.

```
// Parse a URL
const url = require('url');
const myUrl = new URL('https://example.com:8080/
path?name=node&lang=js');
console.log(myUrl.hostname);   // example.com
console.log(myUrl.port);       // 8080
console.log(myUrl.searchParams.get('name')); // node
```

6 events

Event Emitter

Create and work with events and event listeners.

```
// Create an event
const EventEmitter = require('events');
const eventl = new EventEmitter();
event.on('greet', (name) => {
  console.log('Hello ' + name);
});
event.emit('greet', 'Node.js');
```

7 crypto

Crypto

Perform cryptographic operations like hashing, encryption, and more.

```
// Create hash
const crypto = require('crypto');
const hash = crypto.createHash('sha256')
 .update('Hello Node.js')
 .digest('hex');
console.log(hash);
```

8 stream

Stream

Work with streams. Read or write data piece by piece.

```
// Readable stream
const fs = require('fs');
const readable = fs.createReadStream('file.txt',
  { encoding: 'utf8' });
readable.on('data', (chunk) => {
  console.log('Received:', chunk);
});
```

Built-in & Reliable
Core modules are tested and maintained by Node.js team.

No Extra Installation
Use these modules right out of the box.

High Performance
Optimized for speed and efficiency.

Cross Platform
Works the same on Windows, macOS, Linux, and more.

Developer Friendly
Simple, powerful, and easy to use.

npm Basics

Node Package Manager – Install, Manage & Share Packages

World's Largest Package Registry — 1M+ packages

Easy to Use — Simple commands

Reliable & Secure — Verified publishers and integrity checks

Open Source — Free & community driven

1 npm

npm is the default package manager for Node.js. It helps you discover, install and manage packages.

```
$ npm -v      # Check npm version
$ node -v     # Check Node.js version
```

2 package.json

Manifest file for your project. Contains metadata, dependencies, scripts and more.

```
{
  "name": "my-project",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "start": "node index.js"
  },
  "dependencies": {
    "express": "^4.18.2"
} }
```

3 package-lock.json

Automatically generated file that locks the exact versions of installed packages and their dependencies.

```
{
  "name": "my-project",
  "lockfileVersion": 3,
  " packages": {
    "": {
      "dependencies": {
        "express": "4.18.2"
      }
  }
}
```

4 npm init

Initialize a new npm project and create a package.json file.

```
$ npm init
$ npm init -y   # Create with defaults
```

Creates a basic package.json

```
{
  "name": "my-project",
  "version": "1.0.0",
  "main": "index.js",
  "license": "MIT"
}
```

5 npm install

Install packages and save them in dependencies.

```
$ npm install <package>      # Install
$ npm install                # Install all
$ npm install <package> -D   # Dev dependency
$ npm install --save-exact   # Exact version
```

Example

```
$ npm install express
$ npm install nodemon -D
```

6 npm uninstall

Uninstall a package from your project.

```
$ npm uninstall <package>      # Remove
$ npm uninstall <package> -D   # Remove dev dep
```

Example

```
$ npm uninstall express
$ npm uninstall nodemon -D
```

7 Dependencies

Packages your project needs to run.

Types of Dependencies

- dependencies – Required for production
- devDependencies – Required for development
- peerDependencies – For plugins and libraries
- optionalDependencies – Optional packages

8 Scripts

Run custom commands using npm scripts.

```
"scripts": {
  "start": "node index.js",
  "dev": "nodemon index.js",
  "build": "node build.js",
  "test": "jest"
}
```

```
$ npm run <script>
$ npm run dev      # Run dev script
$ npm start        # Run start script
```

Why npm?

Discover — Find amazing packages built by the community.

Install — One command to install and update packages.

Manage — Easily manage dependencies and versions.

Automate — Use scripts to automate tasks and workflows.

Share — Publish your packages and help others.

Node.js Package Management

Manage dependencies, versions & packages like a pro

Organize — Keep your project clean & structured

Reliable — Use trusted packages

Fast — Speed up development

Community — Leverage the power of open source

1 Dependencies

Packages your project needs in production.

```
$ npm install express
$ npm install cors
$ npm install mongoose
```

Saved in package.json under "dependencies"

2 Dev Dependencies

Packages needed only for development & testing.

```
$ npm install --save-dev nodemon
$ npm install --save-dev jest
$ npm install --save-dev eslint
```

Saved in package.json under "devDependencies"

3 Semantic Versioning

Version format: **MAJOR.MINOR.PATCH**
Example: `1.2.3`

1	2	3
MAJOR	MINOR	PATCH
Breaking changes	New features (backward compatible)	Bug fixes (backward compatible)

Use ^, ~, or exact versions to control updates.

`^1.2.3` `~1.2.3` `1.2.3`

4 package.json

Manifest file that holds project info & dependency list.

```
{
  "name": "my-project",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "test": "jest"
  },
  "dependencies": {
    "express": "^4.18.2"
  },
  "devDependencies": {
    "nodemon": "^3.1.0"
  }
}
```

Key Fields

- name
- version
- main
- scripts
- dependencies
- devDependencies
- engines

node_modules

Folder where installed packages are stored.

```
node_modules/
├── express/
│   ├── index.js
│   ├── package.json
│   └── ...
├── mongoose/
├── cors/
└── ...
```

- ✓ Auto-generated
- ✓ Don't edit manually
- ✓ Add to .gitignore
- ✓ Contains all installed packages

npm update

Update packages to the latest compatible versions.

```
$ npm update           # Update ail packages
$ npm update express   # Update specific
$ npm outdated         # Check outdated
                         packages
```

Tip
Use ^ or ~ in package.json to get safe updates automatically.

npm audit

Scan your project for vulnerabilities in dependencies.

```
$ npm audit            # Run audit
$ npm audit fix        # Fix issues
$ npm audit fix --force # Force fix
```

Security First
Always run npm audit regularly and keep your packages updated.

npm cache

Manage npm cache to improve performance.

```
$ npm cache verify     # Verify cache
$ npm cache clean --force  # Clear cache
$ npm cache ls         # List cache
```

Why Cache?
Cache makes installs faster by reusing previously downloaded packages.

Quick Summary

Install packages `npm install <package>`	Install dev packages `npm install -D <package>`	Update packages `npm update`	Check security `npm audit`	Manage cache `npm cache clean --force`	Project config `package.json`	Installed packages `node_modules`

Best Practice: Keep dependencies updated, remove unused packages, and always check for vulnerabilities.

Pro Tip: Use .npmrc to configure registry, cache location, and other npm settings.

Node.js File System (FS)

Work with files and folders easily using the built-in fs module

Built-in Module — No extra install required

Fast & Efficient — High performance I/O operations

Asynchronous — Non-blocking by default

Cross Platform — Works on Windows, macOS, Linux

1 readFile

Read data from a file.

Callback

```
fs.readFile('file.txt', 'utf8',
  (err, data) => {
   if (err) throw err;
   console.log(data);
});
```

Sync

```
const data = fs.readFileSync('file.txt', 'utf8');
```

2 writeFile

Write data to a file.

Callback

```
fs.writeFile('file.txt', 'Hello Node.js!',
  (err) => {
   if (err) throw err;
   console.log('File written!');
});
```

Sync

```
fs.writeFileSync('file.txt', 'Hello Node.js!');
```

3 appendFile

Append data to the end of a file.

Callback

```
fs.appendFile('file.txt', '\nAppended text',
  (err) => {
   if (err) throw err;
   console.log('Data appended!');
});
```

Sync

```
fs.appendFileSync('file.txt', '\nAppended text');
```

4 rename

Rename or move a file or folder.

Callback

```
fs.rename('old.txt', 'new.txt', (err) => {
  if (err) throw err;
  console.log('Renamed!');
});
```

Sync

```
fs.renameSync('old.txt', 'new.txt');
```

5 unlink

Delete a file.

Callback

```
fs.unlink('file.txt', (err) => {
  if (err) throw err;
  console.log('File deleted!');
});
```

Sync

```
fs.unlinkSync('file.txt');
```

6 mkdir

Create a new directory.

Callback

```
fs.mkdir('myFolder', { recursive: true },
  (err) => {
   if (err) throw err;
   console.log('Folder created!');
});
```

Sync

```
fs.mkdirSync('myFolder', { recursive: true });
```

7 readdir

Read files and folders in a directory.

Callback

```
fs.readdir('.', (err, files) => {
  if (err) throw err;
  console.log(files);
});
```

Sync

```
const files = fs.readdirSync('.');
```

8 Other Helpful Methods

- **stat() / statSync()** – Get file/directory info
- **existsSync()** – Check if path exists
- **copyFile() / copyFileSync()** – Copy file
- **rmdir() / rmdirSync()** – Remove directory
- **watch()** – Watch for changes
- **readFileSync()** – Read file (synchronous)
- **writeFileSync()** – Write file (synchronous)

9 Promises API (fs.promises)

Use modern Promise-based API (Node.js 10.0.0+).

```
const fs = require('fs').promises;
```

Read File

```
const data = await fs.readFile('file.txt', 'utf8');
console.log(data);
```

Write File

```
await fs.writeFile('file.txt', 'Hello Node.js!');
```

Append File

```
await fs.appendFile('file.txt', '\nAppended text');
```

Create Folder

```
await fs.mkdir('myFolder', { recursive: true });
```

Read Directory

```
const files = await fs.readdir('.');
console.log(files);
```

Delete File

```
await fs.unlink('file.txt');
```

Promises make code cleaner with async/await.

Common Encoding

- **utf8** – Most common (text)
- **ascii** – ASCII character set
- **base64** – Base64 encoded
- **binary** – Binary data

Example: Read, Modify & Write

```
const fs = require('fs');
fs.readFile('note.txt', 'utf8', (err, data) => {
  if (err) throw err;
  const newData = data + '\nUpdated using Node.js!';
  fs.writeFile('note.txt', newData, (err) => {
    if (err) throw err;
    console.log('File updated successfully!');
  });
});
```

Always handle errors and use async methods to keep your app fast and non-blocking.

Best Practices

- ✓ Use asynchronous methods (fs.promises) for better performance.
- ✓ Always handle errors with try/catch or callbacks.
- ✓ Validate file paths and user input.
- ✓ Use { recursive: true } for nested directories.
- ✓ Keep your I/O operations non-blocking for scalability.

Node.js Path Module

Work with file and directory paths like a pro

Cross Platform — Works on Windows, macOS, Linux

Built-in Module — No installation required

Utility First — Makes path handling easier

Developer Friendly — Simple, consistent and reliable

1 join()

Joins all given path segments together using the correct separator.

```
const path = require('path');
const p = path.join('folder', 'sub', 'file.txt');
console.log(p);
```

Output: folder/sub/file.txt (Unix)
folder\sub\file.txt (Windows)

2 resolve()

Resolves a sequence of paths or path segments into an absolute path.

```
const path = require('path');
const p = path.resolve('folder', 'sub', 'file.txt');
console.log(p);
```

Output: /home/user/folder/sub/file.txt (Unix)
C:\Users\User\folder\sub\file.txt (Windows)

3 basename()

Returns the last portion of a path. (By default it returns the file name)

```
const path = require('path');
const p = path.basename('folder/sub/file.txt');
console.log(p);
```

Output: file.txt

4 dirname()

Returns the directory name of a path.

```
const path = require('path');
const p = path.dirname('/folder/sub/file.txt');
console.log(p);
```

Output: /folder/sub

5 extname()

Returns the extension of the path, including the dos (.).

```
const path = require('path');
const p = path.extname('/folder/sub/file.txt');
console.log(p);
```

Output: .txt

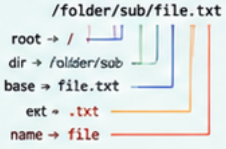
6 parse()

Returns an object with details about the path.

```
const path = require('path');
const p = path.parse('/folder/sub/file.txt');
console.log(p);
```

Output:

```
{
  root: '/',
  dir: '/folder/sub',
  base: 'file.txt',
  ext: '.txt',
  name: 'file'
}
```



7 format()

Returns a path string from an object produced by path.parse().

```
const path = require('path');
const p = path.format({
  dir: '/folder/sub',
  name: 'file',
  ext: '.txt'
});
console.log(p);
```

Output: /folder/sub/file.txt

8 normalize()

Normalizes a path, resolving .. and . segments.

```
const path = require('path');
const p = path.normalize('/folder//sub/../file.txt');
console.log(p);
```

Output: /folder/file.txt (Unix)
\folder\file.txt (Windows)

Quick Tips

- Always use path.join() to build paths.
- Use path.resolve() to get absolute paths.
- path.basename() and path.dirname() are very useful for file handling.
- path.normalize() helps clean up messy paths.

Example: Combine Multiple Methods

```
const path = require('path');
const filePath = path.join('assets', 'images', 'logo.png');
console.log('Joined:', filePath);
console.log('Dirname:', path.dirname(filePath));
console.log('Extension:', path.extname(filePath));
console.log('Basename:', path.basename(filePath));
```

```
Output:
Joined: assets/images/logo.png
Dirname: assets/images
Extension: .png
Basename: logo.png
```

Node.js OS Module

Get operating system information & utilities

Built-in Module — No installation required

Cross Platform — Works on Windows, macOS, Linux

Reliable — Accurate system information

Developer Friendly — Simple API for system details

1 hostname()

Returns the hostname of the operating system.

```
const os = require('os');
console.log(os.hostname());
```

```
Output: my-pc                    // (string)
```

2 platform()

Returns the operating system platform.

```
const os = require('os');
console.log(os.platform());
```

```
Output: win32
                    // e.g. win32, linux, darwin
```

3 arch()

Returns the CPU architecture of the system.

```
const os = require('os');
console.log(os.arch());
```

```
Output: x64
                    // e.g. x64, ia32, arm64
```

4 cpus()

Returns an array of CPU information objects.

```
const os = require('os');
console.log(os.cpus());
```

```
Output: [ { model: 'Intel(R) Core(TM) i5-..',
        speed: 2300, times: { ... } }, ... ]
        // (array)
```

5 freemem()

Returns the amount of free system memory in bytes.

```
const os = require('os');
console.log(os.freemem());
```

```
Output: 4301936640
                    // (number - bytes)
```

6 totalmem()

Returns the total system memory in bytes.

```
const os = require('os');
console.log(os.totalmem());
```

```
Output: 8589934592
                    // (number - bytes)
```

7 uptime()

Returns the system uptime in seconds.

```
const os = require('os');
console.log(os.uptime());
```

```
Output: 123456.78
                    // (number - seconds)
```

8 userinfo()

Returns information about the current user.

```
const os = require('os');
console.log(os.userInfo());
```

```
Output: { uid: -1, gid: -1,
        username: 'User',
        homedir: 'C:\\Users\\User', shell: null }
        // (object)
```

How to Use

1. Import the os module

```
const os = require('os');
```

2. Call any method

```
os.platform();
```

Quick System Info Example

```
const os = require('os');
console.log('Hostname   :', os.hostname());
console.log('Platform   :', os.platform());
console.log('Arch       :', os.arch());
console.log('CPUs       :', os.cpus().length);
console.log('Free Mem   :', os.freemem());
console.log('Total Mem  :', os.totalmem());
console.log('Uptime     :', os.uptime(), 'seconds');
console.log('User Info  :', os.userInfo());
```

```
Hostname   : my-pc
Platform   : win32
Arch       : x64
CPUs       : 8
Free Mem   : 4301936640
Total Mem  : 8589934592
Uptime     : 123456.78 seconds
User Info  : { uid: -1, gid: -1, username: 'User' }
```

Pro Tips

- Use os.cpus()..length to get number of CPU cores.
- Memory values are in bytes. Convert using custom function (e.g., bytes to MB/GB).
- All methods are synchronous and fast.
- Great for system monitoring & logging.

Use Cases: System monitoring · Logging system info · Performance analysis · Environment checks · DevOps & Automation

Note: All methods are synchronous and part of Node.js core.

Node.js HTTP Module

Build web servers and handle requests like a pro

Built-in Module — No extra installation

Fast & Lightweight — High performance I/O operations

Flexible — Full control over requests & responses

Cross Platform — Works on Windows, macOS, Linux

1 createServer()

Creates an HTTP server. It takes a callback function with (req, res) as parameters.

```
const http = require('http');
const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello from Node.js!\n');
});
server.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

```
Output: Server running on port 3000
```

2 Request

The req (request) object contains information about the incoming HTTP request.

Useful Properties

- `req.method` – GET, POST, PUT, DELETE, etc.
- `req.url` – URL of the request
- `req.headers` – Request headers
- `req.query` – Query string (if any)
- `req.body` – Request body (for POST/PUT)
- `req.httpVersion` – HTTP version

```
console.log(req.method);     // GET
console.log(req.url);        // /api/data
console.log(req.headers);    // { host: 'localhost:3000', ... }
```

3 Response

The res (response) object is used to send data back to the client.

Useful Methods

- `res.writeHead(statusCode, headers)` – Set status & headers
- `res.write(data)` – Write data to response
- `res.end(data)` – End the response
- `res.setHeader(name, value)` – Set header
- `res.statusCode` – Get/Set status code

```
res.writeHead(200, { 'Content-Type': 'text/plain' });
res.write('Hello, Client!');
res.end();
```

4 Headers

Headers contain important metadata about the request or response.

Common Response Headers

- `Content-Type` – Type of content
- `Content-Length` – Size of content
- `Cache-Control` – Cache control
- `Set-Cookie` – Set cookies

```
res.setHeader('Content-Type', 'application/json');
res.setHeader('Cache-Control', 'no-cache');
```

5 Status Code

Status codes tell the client the result of the request.

Code	Meaning	Description
200	OK	Request successful
201	Created	Resource created
400	Bad Request	Invalid request
404	Not Found	Resource not found
500	Internal Server Error	Server error

```
res.writeHead(404, { 'Content-Type': 'text/plain' });
res.end('404 Not Found');
```

6 Routing

Route requests based on URL and HTTP method.

```
const server = http.createServer((req, res) => {
  if (req.url === '/') {
    res.end('Home Page');
  } else if (req.url === '/about') {
    res.end('About Page');
  } else {
    res.writeHead(404);
    res.end('Page Not Found');
  }
});
```

```
Supports: GET, POST, PUT, DELETE, etc.
```

7 JSON Response

Send JSON data to the client easily.

```
const data = {
  message: 'Success',
  status: 200,
  data: [1, 2, 3]
};
res.writeHead(200, {
  'Content-Type': 'application/json'
});
res.end(JSON.stringify(data));
```

```
Output (JSON):
{ "message": "Success", "status": 200, "data": [1, 2, 3] }
```

8 Port

Start the server on a specific port to listen for requests.

```
const PORT = 3000;
server.listen(PORT, () => {
  console.log('Server is running on port ${PORT}');
});
```

Common Ports

- `80` – HTTP (Web)
- `443` – HTTPS (Secure)
- `3000` – Development (Node.js default)

How HTTP Works

Best Practices

- Always set appropriate status codes.
- Set Content-Type header correctly.
- Handle errors and invalid routes.
- Keep your server lightweight and fast.

Tip

Use Express.js for more advanced features like middleware, routing, and error handling.

```
npm install express
```

Node.js URL Module

Work with URLs and URLSearchParams easily

Built-in Module — No extra installation

Fast & Reliable — High performance URL handling

Standards Based — WHATWG URL standard

Cross Platform — Works on Windows, macOS, Linux

1 URL

The URL class parses and formats URLs.

Create a new URL object.

```
const { URL } = require('url');
const myURL = new URL('https://user:pass@
www.example.com:8080/path/name?query=1&
page=2#section');
```

Full URL Example

```
https://user:pass@www.example.com:8080/
path/name?query=1&page=2#section
```

2 URL Properties

Access different parts of a URL.

Property	Description	Example Value
protocol	Protocol of the URL	https:
username	Username in the URL	user
password	Password in the URL	pass
hostname	Hostname of the URL	www.example.com
port	Port number	8080
pathname	Path of the URL	/path/name
search	Query string (with ?)	?query=1&page=2
hash	Hash (with #)	#section
origin	Origin (protocol + host + port)	https://www.example.com:8080

3 URLSearchParams

Easily work with query parameters.

```
const { URL, URLSearchParams } = require('url');
const myURL = new URL('https://example.com/search?name=node&
lang=js&page=2');
const params = myURL.searchParams;
```

Common Methods

Method	Description	Example
get(name)	Get value of a parameter	params.get('name') // 'node'
set(name, value)	Set a parameter	params.set('page', '3')
append(name, value)	Add a new parameter	params.append('sort', 'asc')
has(name)	Check if parameter exists	params.has('lang') // true
delete(name)	Delete a parameter	params.delete('lang')
toString()	Get query string	params.toString()

4 Protocol

Includes the trailing ':' (e.g., 'https:')

```
myURL.protocol;
// 'https:'
```

5 Hostname

Returns the domain name (e.g., 'www.example.com')

```
myURL.hostname;
// 'www.example.com'
```

6 Pathname

Returns the path after the domain (e.g., '/path/name')

```
myURL.pathname;
// '/path/name'
```

7 Query

Returns the query string including '?' (e.g., '?query=1&page=2')

```
myURL.search;
// '?query=1&page=2'
```

8 Hash

Returns the hash including '#' (e.g., '#section')

```
myURL.hash;
// '#section'
```

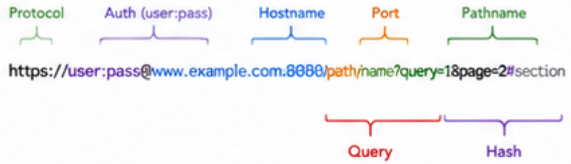
9 Origin

Returns the origin (protocol + hostname + port) (e.g., 'https://www.example.com:8080')

```
myURL.origin;
// 'https://www.example.com:8080'
```

Example: Putting it All Together

```
const { URL } = require('url');
const myURL = new URL('https://user:pass@www.example.com:8080/path/name?query=1&page=2#section');

console.log(myURL.protocol);     // https:
console.log(myURL.hostname);     // www.example.com
console.log(myURL.pathname);     // /path/name
console.log(myURL.search);       // ?query=1&page=2
console.log(myURL.hash);         // #section
console.log(myURL.origin);       // https://www.example.com:8080
```



Best Practices

- Use the URL class to parse and format URLs.
- Use URLSearchParams for easy query string manipulation.
- Always validate and sanitize user inputs in URLs.
- Prefer the built-in 'url' module over manual string parsing.

Quick Tip

URLSearchParams automatically decodes and encodes query parameters for you!

```
params.set('q', 'hello world');
// q=hello+world
```

Pro Tip: The WHATWG URL standard is followed in Node.js, ensuring consistent behavior across platforms.

Learn More: https://nodejs.org/api/url.html

Node.js Events Module

Handle events & build event-driven applications

Built-in Module — No extra installation

Lightweight — Fast and efficient

Core Feature — Used in streams, http, fs, etc.

Event Driven — Build scalable & reactive apps

1 EventEmitter

The core of the Events module. Used to create, emit & listen to events.

```
const EventEmitter = require('events');
const emitter = new EventEmitter();

console.log(emitter instanceof EventEmitter);
// true
```

Almost all event-driven objects in Node.js are instances of EventEmitter.

2 emit()

Triggers an event and calls all listeners.

```
emitter.on('greet', (name) => {
  console.log('Hello, ${name}!');
});

emitter.emit('greet', 'Node.js');
```

Output:

```
Hello, Node.js!
```

3 on()

Registers a listener for an event. Can be called multiple times.

```
emitter.on('data', (data) => {
  console.log('Data:', data);
});

emitter.emit('data', 'Any Data');
```

Output:

```
Data: Any Data
```

4 once()

Registers a listener that will be invoked only once.

```
emitter.once('start', () => {
  console.log('Started (only once)');
});
emitter.emit('start');
emitter.emit('start');
```

Output:

```
Started (only once)
```

5 removeListener()

Removes a specific listener from an event.

```
const handler = () => console.log('Bye!');
emitter.on('end', handler);
emitter.emit('end');  // Bye!
emitter.removeListener('end', handler);
emitter.emit('end');  // (no output)
```

Output:

```
Bye!
```

6 Custom Events

You can create and work with your own custom events.

```
emitter.on('login', (user) => {
  console.log('${user} logged in');
});
emitter.emit('login', 'John');
```

Output:

```
John logged in
```

7 Listeners

Get the number of listeners for an event or all event names.

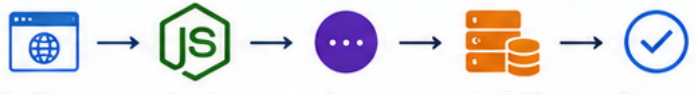
```
emitter.on('msg', () => {});
emitter.on('msg', () => {});
console.log(emitter.listenerCount('msg')); // 2
console.log(emitter.eventNames());         // [ 'msg' ]
```

Useful Methods

- `emitter.listenerCount(event)` – Returns number of listeners
- `emitter.eventNames()` – Returns array of event names
- `emitter.listeners(event)` – Returns array of listeners

8 Event Driven

Node.js uses an event-driven, non-blocking I/O model.



Benefits

- Handles many concurrent connections
- No thread blocking
- High performance & scalable

Use Cases

- Real-time chat applications
- Live notifications
- Streaming data
- File watching
- HTTP requests/responses
- Any task with async events

Quick Example

```
const EventEmitter = require('events');
const emitter = new EventEmitter();

emitter.on('order', (id) => console.log('Order ${id} placed'));
emitter.emit('order', 101);
emitter.emit('order', 102);
```

Output:

```
Order 101 placed
Order 102 placed
```

Best Practices

- Use meaningful event names.
- Avoid too many listeners on the same event.
- Remove listeners when they are no longer needed.
- Use once() for events that should trigger only once.
- Keep your event-driven code clean and modular.

Pro Tip

Node.js core modules like http, fs, stream, and net are built using EventEmitter internally.

```
const http = require('http');
const server = http.createServer((req, res) => {
  res.end('Hello!');
});
server.listen(3000);
```

Event Driven

Node.js Streams

Process data efficiently with Streams

Memory Efficient — Handle large data without loading everything in memory

High Performance — Ideal for I/O intensive operations

Built-in Support — Core module stream

Event Driven — Non-blocking and scalable

1 Readable

Streams from which data can be read.

```
const fs = require('fs');
const readableStream = fs.createReadStream('file.txt');
readableStream.on('data', (chunk) => {
  console.log(chunk.toString());
});
```

Use Cases:
- Reading files
- HTTP requests
- Database responses

2 Writable

Streams to which data can be written.

```
const fs = require('fs');
const writableStream = fs.createWriteStream('file.txt');
writableStream.write('Hello Node.js Streams!');
writableStream.end();
```

Use Cases:
- Writing files
- HTTP responses
- Sending data to sockets

3 Duplex

Streams that are both Readable and Writable.

```
const net = require('net');
const socket = net.createConnection(3000, 'localhost');
socket.write('Hey Server!');
socket.on('data', (data) => {
  console.log(data.toString());
});
```

Use Cases:
- TCP sockets
- Chat applications
- Network connections

4 Transform

Duplex streams that modify or transform the data.

```
const { Transform } = require('stream');
const upperCase = new Transform({
  transform(chunk, encoding, callback) {
    callback(null, chunk.toString().toUpperCase());
  }
});
```

Use Cases:
- Data compression
- Encryption / Decryption
- Formatting & parsing

5 Pipe

Connects the output of one stream to the input of another.

```
const fs = require('fs');
const read = fs.createReadStream('input.txt');
const write = fs.createWriteStream('output.txt');
read.pipe(write);
```

Pipe handles data flow, backpressure and errors automatically.

6 Buffer

A temporary storage that holds binary data.

```
const buf = Buffer.from('Hello');
console.log(buf);         // <Buffer 48 65 6c 6c 6f>
console.log(buf.toString()); // Hello
```

Buffers are used internally by streams to handle binary data.

7 Chunk

A small piece of data read from or written to a stream.

```
readableStream.on('data', (chunk) => {
  console.log('Chunk received:', chunk.length, 'bytes');
});
```

Streams work with chunks, not the whole data at once.

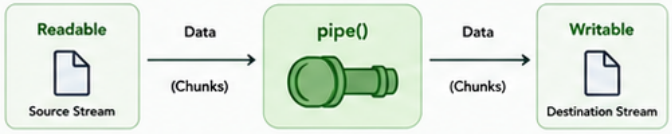
8 Streaming

Processing data in chunks as it is generated.

```
// Ideal for large files, live data,
// real-time apps, and low memory usage
```

Streaming = Faster, Scalable, Memory Efficient!

How Pipe Works



Pipe manages backpressure and ensures smooth data flow.

Quick Example

```
const fs = require('fs');
const readStream = fs.createReadStream('big-file.mp4');
const writeStream = fs.createWriteStream('copy.mp4');
readStream.on('data', (chunk) => {
  console.log('Received chunk: ${chunk.length} bytes');
});
readStream.on('end', () => {
  console.log('Finished reading!');
});
readStream.pipe(writeStream);
```

Output (Console)

```
Received chunk: 65536 bytes
Received chunk: 65536 bytes
Received chunk: 65536 bytes
...
Finished reading!
```

Best Practices

- Always handle 'error' and 'end' events.
- Use pipe() for simple stream piping.
- Use appropriate highWaterMark for performance.
- Do not load large files in memory, use streams.

Pro Tip

Streams are the key to building fast, scalable and memory-efficient Node.js apps!

Real World Use Cases: File uploads | Video & audio streaming | Log processing | Real-time chat | Data compression

Remember: Streams are event-driven, non-blocking and built for scale.

Node.js Buffers

Work with binary data efficiently

Binary Safe — Store any type of binary data

High Performance — Optimized for speed and memory

Built-in Support — Core Node.js module

Versatile — Encoding, decoding & conversions

1 Buffer

Buffer is a global object used to handle binary data.

```
const buf = Buffer.from('Hello');
console.log(buf);
// <Buffer 48 65 6c 6c 6f>
```

A Buffer is similar to an array of integers representing bytes.

2 Binary Data

FF 00 1A 7F

Buffers store data as raw bytes (0-255).

```
const buf = Buffer.from([72, 101, 108, 108, 111]);
console.log(buf);
// <Buffer 48 65 6c 6c 6f>
```

Each number represents a byte.
Example: 72 -> 'H', 101 -> 'e'

3 alloc()

Allocates a new Buffer with the specified size.

```
const buf = Buffer.alloc(5);
console.log(buf);
// <Buffer 00 00 00 00 00>
```

By default, alloc() fills the buffer with 0.
Use allocUnsafe() for better performance.

4 from()

Creates a Buffer from string, array, or another Buffer.

```
const buf1 = Buffer.from('Node.js');
const buf2 = Buffer.from([78, 111, 100, 101]);
const buf3 = Buffer.from(buf1);
```

Supports: String, Array, Buffer, ArrayBuffer, TypedArray, DataView, etc.

5 toString()

Converts a Buffer to a string using specified encoding.

```
const buf = Buffer.from('Hello');
console.log(buf.toString());        // Hello
console.log(buf.toString('hex'));   // 48656c6c6f
```

Default encoding is 'utf8'.

6 Encoding

Encoding defines how bytes are converted to/from strings.

Encoding	Description
utf8	Unicode Transformation Format (8-bit)
ascii	ASCII encoding
hex	Hexadecimal encoding
base64	Base64 encoding
latin1	Latin-1 encoding

7 Base64

BASE 64

Base64 encoding is commonly used to represent binary data in text.

```
const buf = Buffer.from('Node.js');
const b64 = buf.toString('base64');
console.log(b64);      // Tm9kZS5qcw==
```

Use 'base64' to encode/decode binary data for transmission or storage.

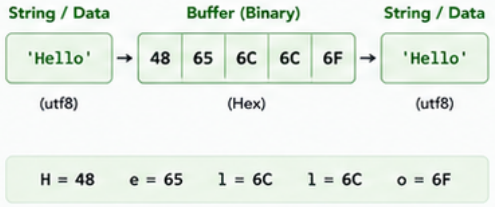
8 UTF-8

UTF-8 is the default encoding for strings in Node.js.

```
const buf = Buffer.from('Hello 👋', 'utf8');
console.log(buf);          // <Buffer 48 65 6c 6c 6f
                                    f0 9f 91 8b>
```

UTF-8 supports all Unicode characters and is ideal for international text.

How Buffer Works (Visual)



H = 48 e = 65 l = 6C l = 6C o = 6F

Common Examples

- Create from string: `Buffer.from('Hello', 'utf8')`
- Create from array: `Buffer.from([72,101,108,108,111])`
- Allocate buffer: `Buffer.alloc(10)`
- Convert to string: `buf.toString('utf8')`
- Convert to Base64: `buf.toString('base64')`

Sample Code

```
const buf = Buffer.from('Node.js');
console.log(buf);                       // <Buffer 4e 6f 64 65 2e 6a 73>
console.log(buf.length);                // 7
console.log(buf.toString());  //);      // Node.js
console.log(buf.toString('hex'));       // 4e6f646S2e6a73
console.log(buf.toString('base64'));    // Tm9kZSSqcw==
```

Output

```
<Buffer 4e 6f 64 65 2e 6a 73>
7
Node.js
4e6f64652e6a73
Tm9kZSSqcw==
```

Best Practices

- Use Buffer.alloc() to avoid uninitialized memory.
- Prefer 'utf8' encoding for text data.
- Use Base64 for safe text transmission of binary data.
- Always handle encoding/decoding correctly.
- Be mindful of buffer size for performance.

Buffers are the foundation for handling binary data in Node.js — from files and streams to networks!

Buffers make Node.js powerful for working with files, streams, networks, images, videos, and much more! | **Remember:** Everything in a Buffer is just bytes. | **Learn more:** https://nodejs.org/api/buffer.html

Node.js Timers

Schedule & manage async tasks with timers

Event Loop Friendly — Timers run in the Event Loop

Non-Blocking — Does not block code execution

High Performance — Efficient for scheduling tasks

Flexible — Great for delays, repeats & queues

1 setTimeout()

Executes a function once after a specified delay.

```
// Run once after 2 seconds
setTimeout(() => {
  console.log('Timeout executed!');
}, 2000);

console.log('This runs first!');
```

Output Order:

1. This runs first!
2. Timeout executed! (after 2 sec)

2 setInterval()

Executes a function repeatedly with a fixed interval.

```
let count = 0;
const id = setInterval(() => {
  count++;
  console.log('Interval:', count);
  if (count === 3) clearInterval(id);
}, 1000);
```

Output (every 1 sec):

```
Interval: 1    Interval: 2    Interval: 3
```

3 setImmediate()

Executes the callback in the check phase of the event loop.

```
setImmediate(() => {
  console.log('Immediate executed!');
});

console.log('This runs first!');
```

Output Order:

1. This runs first!
2. Immediate executed!

4 clearTimeout()

Cancels a scheduled setTimeout callback.

```
const timer = setTimeout(() => {
  console.log('You will not see this!');
}, 3000);

clearTimeout(timer);
console.log('Timeout cleared!');
```

Output:

```
Timeout cleared!
```

5 clearInterval()

Stops a setInterval from running.

```
const id = setInterval(() => {
  console.log('Tick...');
}, 1000);

setTimeout(() => {
  clearInterval(id);
  console.log('Interval cleared!');
}, 3500);
```

Output (every 1 sec for ~3 sec):

```
Tick...
Tick...
Interval cleared!
```

6 process.nextTick()

Executes the callback on the next iteration of the event loop, before I/O events.

```
console.log('Start');
process.nextTick(() => {
  console.log('nextTick callback');
});
console.log('End');
```

Output Order:

1. Start
2. End
3. nextTick callback

7 Async Tasks

Timers are asynchronous and won't block code execution.

```
console.log('A');
setTimeout(() => console.log('B'), 2000);
console.log('C');
```

Explanation:

A and C run immediately.
B runs after 2 seconds.

8 Scheduling

Use timers to schedule delayed or repeated tasks easily.

```
// Run a task every minute
setInterval(() => {
  console.log('Task executed every minute');
}, 60 * 1000);

// Run once after 10 seconds
setTimeout(() => {
  console.log('Task executed after 10 sec');
}, 10000);
```

Useful for reminders, polling, auto-save, notifications, and more.

Event Loop Phases (Where Timers Fit)

Timers callbacks run in the `'timers'` phase.

Key Points

- `setTimeout` / `setInterval` run in the timers phase.
- `setImmediate` runs in the check phase.
- `process.nextTick` runs before I/O and before other phases.

Quick Comparison

Method	Type	When it runs	Use For
setTimeout()	One-time	After delay (ms) in timers phase	Delayed execution
setInterval()	Repeated	Every interval (ms) in timers phase	Repeating tasks
setImmediate()	One-time	In check phase (after poll)	After I/O, immediate tasks
process.nextTick()	One-time	Before event loop continues	High priority tasks
clearTimeout()	Cancel	Cancels setTimeout	Stop one-time timer
clearInterval()	Cancel	Cancels setInterval	Stop repeating timer

Best Practices

- Always clear timers when no longer needed.
- Avoid long-running operations inside timer callbacks.
- Use `setImmediate` for tasks after I/O operations.
- Use `process.nextTick` sparingly (can starve I/O).
- Keep timer delays realistic; avoid 0ms loops.

Pro Tip

A delay of `0ms` in `setTimeout(fn, 0)` does not mean "immediate".
It runs after the current call stack is cleared.

```
setTimeout(() => console.log('Later'), 0);
console.log('Now');
```

Output:

```
Now
Later
```

Remember: Timers in Node.js are non-blocking and powered by the Event Loop. Use them smartly to build fast and scalable applications!

Learn more: https://nodejs.org/api/timers.html

Node.js Process Object

Interact with the Node.js process

Built-in Global — Available in every Node.js app

Control & Info — Manage the current Node.js process

Powerful — Essential for CLI tools, servers & scripts

Efficient — Monitor, control & optimize your apps

1 process

The global process object provides information about, and control over, the current Node.js process.

```
// process is global, no need to import
console.log(typeof process);     // object
console.log(process.version);  // v20.x.x
console.log(process.platform); // win32, darwin, linux
```

process is available in every Node.js application.

2 process.argv

An array containing command line arguments passed to the Node.js process.

```
// Run: node app.js one two three
console.log(process.argv);
/*
['node', 'app.js', 'one', 'two', 'three']
*/
```

Example:
`node app.js hello world`

3 process.env

An object containing the user environment.

```
console.log(process.env.NODE_ENV);
console.log(process.env.PATH);
console.log(process.env.HOME);
// Access any environment variable
```

Example:
`PORT=3000 node app.js`

4 process.exit([code])

Ends the process and exits the Node.js application.

```
console.log('Before exit');
process.exit(0);   // 0 = success
// Code below will not run
console.log('This will not run');
```

Exit Codes:
- `0` Success
- `1+` Error / Failure

5 process.cwd()

Returns the current working directory.

```
const cwd = process.cwd();
console.log(cwd);
// Example: C:\Projects\my-app
// or /home/user/my-app
```

Useful for working with files and relative paths.

6 process.pid

Returns the current process identifier (PID).

```
console.log(process.pid);
// Example: 12345
```

Each running process has a unique PID.

7 process.memoryUsage()

Returns memory usage statistics for the process.

```
const mem = process.memoryUsage();
console.log(mem);
/*
{ rss: 12345678, heapTotal: 5678901,
  heapUsed: 3456789, external: 12345 }
*/
```

- `rss` - Resident Set Size
- `heapTotal` - Total heap allocated
- `heapUsed` - Used heap
- `external` - Memory used by C++ objects

8 process.stdout

Standard output stream used for writing output.

```
process.stdout.write('Hello, stdout!\n');
console.log('This goes to stdout');
// You can also pipe it
// process.stdout.pipe(fs.createWriteStream('out.log'))
```

Use stdout for normal output, especially in CLI apps.

Quick Overview

Property / Method	Type	Description	Example
process	Object	Global process object	`process.version`
argv	Array	Command line arguments	`process.argv[2]`
env	Object	Environment variables	`process.env.PORT`
exit(code)	Method	Exit the process	`process.exit(0)`
cwd()	Method	Current working directory	`process.cwd()`
pid	Number	Process ID	`process.pid`
memoryUsage()	Method	Memory usage info	`process.memoryUsage()`
stdout	Stream	Standard output stream	`process.stdout.write()`

Common Use Cases

- ✓ Read command line arguments (argv)
- ✓ Access environment variables (env)
- ✓ Get current directory (cwd)
- ✓ Log and output data (stdout)
- ✓ Monitor memory usage (memoryUsage)
- ✓ Exit process with status code (exit)
- ✓ Manage CLI tools and scripts

Example: CLI App Using Process Object

```
// app.js
console.log('PID:', process.pid);
console.log('Node Version:', process.version);
console.log('Arguments:', process.argv.slice(2));
console.log('Current Dir:', process.cwd());
console.log('Environment:', process.env.NODE_ENV || 'development');
console.log('Memory Usage:', process.memoryUsage());
process.stdout.write('Application started successfully!\n');
```

Run: `node app.js arg1 arg2`

Best Practices

- ✓ Always handle exit codes properly.
- ✓ Don't block the event loop.
- ✓ Use environment variables for configuration.
- ✓ Monitor memory for long-running applications.
- ✓ Write clean output using stdout/stderr.

Pro Tip

Use process.env.NODE_ENV to switch between development and production.

```
if (process.env.NODE_ENV === 'production') {
  console.log('Running in production mode');
} else {
  console.log('Running in development mode');
}
```

Remember: The process object is global and always available.

Learn more: https://nodejs.org/api/process.html

Node.js gives you full control over your application process!

Node.js Async Programming

Write non-blocking code and handle async operations like a pro!

Non-Blocking — Keeps your app fast & responsive

Better Performance — Handles many operations together

Robust Code — Handle errors gracefully

Modern Syntax — Clean, readable and easy to maintain

1 Callback

A function passed as an argument to another function, executed later.

```
fs.readFile('file.txt', 'utf8', (err, data) => {
  if (err) {
    console.error('Error:', err);
    return;
  }
  console.log('Data:', data);
});
```

Oldest way of handling async operations but can lead to "callback hell".

2 Promise

An object representing the eventual completion (or failure) of an async task.

```
const fs = require('fs').promises;
fs.readFile('file.txt', 'utf8')
  .then(data => console.log('Data:', data))
  .catch(err => console.error('Error:', err));
```

Promises make async code cleaner and easier to manage.

3 Async

The async keyword makes a function return a Promise automatically.

```
async function readFileAsync() {
  const fs = require('fs').promises;
  const data = await fs.readFile('file.txt', 'utf8');
  return data;
}
readFileAsync();
```

Always returns a Promise. Use with Await for best results.

4 Await

The await keyword pauses async function execution until the Promise resolves.

```
async function getData() {
  try {
    const data = await fs.readFile('file.txt', 'utf8');
    console.log('Data:', data);
  } catch (err) {
    console.error('Error:', err);
  }
```

Can be used only inside async functions. Makes code look synchronous!

5 Then

Handles the resolved value of a Promise. Chained for multiple operations.

```
fs.readFile('file.txt', 'utf8')
  .then(data => {
    console.log('Data:', data);
    return data.toUpperCase();
  })
  .then(upper => console.log('Upper:', upper));
```

.then() is called when the Promise is resolved.

6 Catch

Handles errors (rejected Promises) in the Promise chain.

```
fs.readFile('file.txt', 'utf8')
  .then(data => console.log('Data:', data))
  .catch(err => {
    console.error('Something went wrong:', err);
  });
```

.catch() catches errors in any part of the chain.

7 Finally

Executes code after Promise is settled (resolved or rejected).

```
fs.readFile('file.txt', 'utf8')
  .then(data => console.log('Data:', data))
  .catch(err => console.error('Error:', err))
  .finally(() => {
    console.log('Operation finished.');
  });
```

.finally() runs no matter what happens.

8 Error Handling

Handle errors using try...catch with async/await for robust code.

```
async function getData() {
  try {
    const data = await fs.readFile('file.txt', 'utf8');
    console.log('Data:', data);
  } catch (err) {
    console.error('Handled Error:', err.message);
  }
```

Always wrap await calls in try...catch.

Async Flow Comparison

Quick Example: Async / Await

```
const fs = require('fs').promises;

async function readData() {
  try {
    const data = await fs.readFile('file.txt', 'utf8');
    console.log('File Content:', data);
  } catch (err) {
    console.error('Error:', err.message);
  } finally {
    console.log('Done reading the file.');
  }
}
readData();
```

Output (Example)

```
File Content: Hello Node.js!
Done reading the file.
```

If Error:

```
Error: ENOENT: no such file
or directory, open 'file.txt'
Done reading the file.
```

Best Practices

- Use async/await for better readability.
- Always handle errors using try...catch.
- Don't forget to handle rejections in Promises.
- Use .finally() for cleanup tasks.
- Avoid callback hell – prefer Promises.
- Keep async functions small and focused.

Remember: Async programming helps Node.js handle many tasks at the same time without blocking.

Performance Boost: More users, faster responses, better experience!

Learn more: https://nodejs.org/en/docs/guides/blocking-vs-non-blocking/

Node.js Error Handling

Write robust, reliable and error-resilient applications

Prevent Crashes — Handle errors gracefully

Improve Debugging — Easier to find and fix issues

Better UX — Show meaningful messages

Production Ready — Keep your app stable & reliable

1 try

Wrap code that might throw an error in try block.

```
try {
  const data = JSON.parse(jsonString);
  console.log('Parsed:', data);
}
catch (err) {}
```

Use try to test a block of code for errors.

2 catch

Catch and handle errors thrown in try block.

```
try {
  riskyOperation();
} catch (err) {
  console.error('Something went wrong:', err.mes
sage);
}
```

catch receives an error object with details about the error.

3 throw

Manually throw an error when something goes wrong.

```
function divide(a, b) {
  if (b === 0) {
    throw new Error('Division by zero');
  }
  return a / b;
}
```

Use throw to create custom or built-in errors and stop execution.

4 Error

Built-in Error object provides error info.

```
const err = new Error('Something failed');
console.log(err.message); // Something failed
console.log(err.name);    // Error
console.log(err.stack);   // Stack trace
```

Error includes message, name, stack and more.

5 Custom Error

Create custom error classes for better clarity.

```
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = 'ValidationError';
  }
}
throw new ValidationError('Invalid input');
```

Custom errors make debugging and handling easier.

6 Unhandled Exception

Catch uncaught synchronous errors globally.

```
process.on('uncaughtException', (err) => {
  console.error('Uncaught Exception:', err);
  // Cleanup & exit
  process.exit(1);
});
```

Use only for cleanup & logging. App should exit after this.

7 Promise Rejection

Handle rejected Promises to avoid crashes.

```
process.on('unhandledRejection', (reason, promise) => {
  console.error('Unhandled Rejection:', reason);
  // Log, cleanup, notify, etc.
});
```

Always handle Promise rejections with .catch() or try/catch.

8 Logging

Log errors with useful details for debugging.

```
function logError(err, context = '') {
  console.error('--- ERROR ---');
  if (context) console.error('Context:', context);
  console.error('Message:', err.message);
  console.error('Stack:', err.stack);
  console.error('Time:', new Date().toISOstring());
}
```

Good logs = Faster debugging + Better issue tracking.

9 Error Handling Best Practices

- Always handle errors at the right level.
- Provide meaningful error messages.
- Don't expose sensitive information to users.
- Use custom errors for business logic.
- Log errors with stack trace and context.
- Handle both sync errors & async rejections.
- Fail gracefully and keep the app stable.

Error Types in Node.js

Error	Description	Example
SyntaxError	Invalid JavaScript syntax	Unexpected token
ReferenceError	Using a variable that is not defined	x is not defined
TypeError	Operation on wrong data type	Cannot read property 'x'
RangeError	Value out of range	Invalid array length
URIError	Bad URI/URL	URI malformed
System Error	OS level errors (ENOENT, EACCES, etc.)	ENOENT: no such file

Example: Complete Error Handling Flow

```
async function readFileSafe(path) {
  try {
    const data = await fs.promises.readFile(path, 'utf8');
    console.log('File content:', data);
  } catch (err) {
    if (err.code === 'ENOENT') {
      throw new Error('File not found');
    }
    logError(err, 'readFileSafe');
  }
}
readFileSafe('data.txt').catch(err => logError(err, 'Top Level'));
```

Possible Output

```
--- ERROR ---
Context: readFileSafe
Message: File not found
Stack:  at ...
Time:  2026-07-24T10:30:00.000Z
--- ERROR ---
Context: Top Level
Message: File not found
...
```

Quick Summary

try	Wrap risky code.
catch	Handle the error.
throw	Throw an error.
Error	Error object with details.
Custom Error	Create your own error classes.
Unhandled Exception	Global sync error handler.
Promise Rejection	Global async error handler.
Logging	Log errors with details.

Pro Tip: A good error message should tell WHAT happened, WHERE it happened and WHY.

Remember: Every error is an opportunity to make your app better!

Learn more: https://nodejs.org/api/errors.html

Node.js Debugging

Find bugs faster, fix smarter! 🐞

Find Issues Fast — Identify bugs and fix quickly

Improve Code Quality — Write reliable and robust applications

Save Time — Better debugging, less guessing

Boost Productivity — Powerful tools for developers

1 console.log()

Print values and track the flow of your code.

```
console.log('Hello Debugging');
console.log('User:', user);
console.log('Value ==', value);
```

Best for quick checks and simple debugging.

2 Debugger Statement

Add `debugger;` statement to pause execution.

```
function add(a, b) {
  const sum = a + b;
  debugger;  // Execution will pause here
  return sum;
}
add(5, 3);
```

Works with Chrome DevTools or VS Code.

3 VS Code Debugger

Powerful debugging experience inside VS Code.

- ✓ Set breakpoints
- ✓ Inspect variables
- ✓ Step Over / Into / Out
- ✓ Watch expressions
- ✓ Call stack & scopes

Use built-in Node.js debugger in VS Code.

4 Node Inspect

Debug your Node.js app from the terminal.

```
# Start debugging
node inspect app.js

# Or with arguments
node inspect app.js arg1 arg2
```

Opens DevTools in Chrome at `chrome://inspect`

5 Breakpoints

Pause code execution at specific points.

- Click in the gutter (left of line) or press F9 in VS Code.
- Execution stops at breakpoint.
- Inspect variables and state.

Helps in inspecting values at critical points.

6 Stack Trace

Shows the call stack when an error occurs.

```
Error: Something went wrong!
    at calculate (app.js:10:15)
    at processData (app.js:25:5)
    at Object.<anonymous> (app.js:30:1)
}
```

Helps trace the path of error in your code.

7 Errors

Handle and inspect errors efficiently.

```
try {
  riskyOperation();
} catch (err) {
  console.error('Error:', err.message);
  console.error(err.stack);
}
```

Always log error message + stack for detailed info.

8 Debug Tools

Useful tools to debug Node.js applications.

- ✓ VS Code Debugger
- ✓ Chrome DevTools
- ✓ Node Inspect
- ✓ Postman (for APIs)
- ✓ Nodemon (auto-restart)

Use the right tool for the right job.

VS Code Debugging Workflow

1. **Set Breakpoints** — Click on gutter or press F9
2. **Start Debugging** — Press F5 or use Debug panel
3. **Inspect & Analyze** — Check variables, call stack, scopes
4. **Step Through Code** — Step Over (F10), Step Into (F11), Step Out (Shift+F11)
5. **Fix & Test** — Fix the issue and test again

Tip: Use breakpoints, watch variables and call stack to understand your code better.

Common Debug Commands (Terminal)

Command	Description
`node inspect app.js`	Start debugger and open DevTools
`node --inspect app.js`	Start debugger with WebSocket URL
`node --inspect-brk app.js`	Break on the first line
`debugger;`	Pause execution at this line
`.list`	List source code (in DevTools console)
`.next`	Step over next line
`.step`	Step into function
`.out`	Step out of current function
`.cont`	Continue execution

Example: Debugging with VS Code

app.js

```
1  function divide(a, b) {
2    if (b === 0) {
3      throw new Error('Division by zero');
4    }
5    return a / b;
6  }
7  console.log(divide(10, 0));
```

Variables

```
a: 10
b: 0
err: Error: Division by zero
```

Call Stack

divide	app.js:3:11
<anonymous>	app.js:7:13
Module._compile	internal/modules/...

Best Practices

- ✓ Use meaningful console messages.
- ✓ Remove console.log() before production.
- ✓ Use breakpoints instead of console.log for complex logic.
- ✓ Always check stack trace for exact error location.
- ✓ Handle errors gracefully.
- ✓ Use environment variables to enable debug mode.

Remember: Good debugging is not about finding faults, it's about understanding your code.

Pro Tip: Use "console.table()" to log arrays and objects in a clean table format.

Useful Resources:

- Node.js Docs (Debugging): https://nodejs.org/en/docs/guides/debugging-getting-started
- VS Code Docs: https://code.visualstudio.com/docs/nodejs/nodejs-debugging

Keep Coding

Debug today, ship better tomorrow! 🚀

Express.js Basics

Fast. Minimal. Unopinionated Web Framework for Node.js

Fast Development
Build APIs & Web Apps super quick

Minimal & Flexible
Unopinionated and easy to customize

Powerful Features
Routing, Middleware, Static Files & more

Perfect for APIs
Ideal for REST APIs and microservices

Express.js
Express.js is a fast, minimalist web framework for Node.js that makes building web apps and APIs simple.

Web Framework
Provides robust features for web and mobile apps, without hiding the core features of Node.js.

app.js
The main file where we create the Express app, configure middleware, routes and start server.

Server
Express runs on top of Node.js HTTP module to create a powerful web server.

Middleware
Functions that run during the request-response cycle. Modify req/res.

API
Define endpoints (routes) and handle HTTP methods and requests.

Build RESTful APIs. JSON in, JSON out!

1 Creating a Simple Server (app.js)

```
const express = require('express');
const app = express();
const PORT = 3000;

app.get('/', (req, res) => {
  res.send('Hello Express! 🚀');
});

app.listen(PORT, () => {
  console.log(`Server running on http://localhost:${PORT}`);
});
```

What's Happening?

- ✓ Create an Express app
- ✓ Define a route for /
- ✓ Send a response
- ✓ Start the server and listen on a port

Run:
```
node app.js
```

2 Middleware Example

```
// Built-in Middleware
app.use(express.json());   // Parse JSON bodies
app.use(express.urlencoded({ extended: true }));

// Custom Middleware
app.use((req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next(); // Pass control to next middleware
});
```

Middleware runs in order, and has access to req, res and next.

3 Routing Example

Method	Route	Description	Example
GET	/	Home Route	app.get('/', handler)
GET	/about	About Page	app.get('/about', handler)
POST	/api/users	Create User	app.post('/api/users', handler)
PUT	/api/users/:id	Update User	app.put('/api/users/:id', handler)
DELETE	/api/users/:id	Delete User	app.delete('/api/users/:id', handler)

4 More Routes

```
app.get('/about', (req, res) => {
  res.send('About Page');
});

app.post('/api/users', (req, res) => {
  res.json({ message: 'User created', data: req.body });
});

app.get('/api/users/:id', (req, res) => {
  res.json({ id: req.params.id, name: 'John Doe' });
});
```

Request Object (req)

- req.params -> Route params
- req.query -> Query string
- req.body -> Request body
- req.headers -> Headers

Response Object (res)

- res.send() -> Send text/html
- res.json() -> Send JSON
- res.status() -> Set status code
- res.end() -> End response

5 Project Structure

```
my-express-app/
  └ node_modules/
  └ app.js
  └ package.json
```

Tip: Use nodemon for auto-restart during development.

```
# Install nodemon
npm install -g nodemon

# Run
nodemon app.js
```

6 Common Middlewares

JSON Parser
`express.json()` – Parse JSON request bodies

URL Encoded
`express.urlencoded()` – Parse form data

Static Files
`express.static('public')` – Serve static files

Logger
`morgan` – HTTP request logger

7 Why Use Express.js?

- ✓ Simplifies routing and middleware
- ✓ Huge ecosystem & community
- ✓ Easy integration with databases & APIs
- ✓ Highly extensible and flexible
- ✓ Perfect for RESTful APIs

Express.js in One Line

> Express.js is a minimal and flexible Node.js web framework that provides a robust set of features for web and mobile applications.

HTTP Methods Quick Reference

GET Read | POST Create | PUT Update | DELETE Delete | PATCH Modify

Remember
Express gives you everything you need to build server-side apps fast and easy!

Learn More
Official Docs: https://expressjs.com/
NPM: https://www.npmjs.com/package/express

Express Routing

Powerful & Flexible Routing for Your Node.js Apps

Fast & Simple — Easy to define routes and APIs

Flexible — Handles all HTTP methods & params

Modular — Use Router for clean code

Scalable — Perfect for small to large apps

1 HTTP Methods in Express

GET	Retrieve data from server	`app.get('/users', (req, res) => { res.send('Get all users'); });`
POST	Create new resource	`app.post('/users', (req, res) => { res.send('User created'); });`
PUT	Update entire resource	`app.put('/users/:id', (req, res) => { res.send('User updated'); });`
PATCH	Update partial resource	`app.patch('/users/:id', (req, res) => { res.send('User patched'); });`
DELETE	Delete resource	`app.delete('/users/:id', (req, res) => { res.send('User deleted'); });`

2 Route Parameters

Route parameters are named URL segments used to capture values.

```
app.get('/users/:id', (req, res) => {
  const userId = req.params.id;
  res.send(`User ID: ${userId}`);
});
```

Example: `GET /users/123`

`req.params.id => "123"`

3 Query Parameters

Query parameters are used to filter, search or paginate.

```
app.get('/search', (req, res) => {
  const { q, page, limit } = req.query;
  res.send(`Search: ${q}, Page: ${page},
    Limit: ${limit}`);
});
```

Example: `GET /search?q=node&page=2&limit=10`

`req.query => { q: 'node', page: '2', limit: '10' }`

4 Example Requests

Method	URL	Description	Example Requests
GET	/users	Get all users	GET /users
GET	/users/1	Get user by id	GET /users/1
POST	/users	Create user	POST /users
PUT	/users/1	Update user	PUT /users/1
PATCH	/users/1	Partial update	PATCH /users/1
DELETE	/users/1	Delete user	DELETE /users/1

5 Router (Modular Routing)

Use express.Router() to organize routes in separate files.

```
project/
  app.js
  routes/
    userRoutes.js
```

app.js

```
const express = require('express');
const app = express();
const userRoutes = require('./routes/userRoutes');
app.use('/api/users', userRoutes);
app.listen(3000, () => console.log('Server running'));
```

routes/userRoutes.js

```
const express = require('express');
const router = express.Router();
router.get('/', (req, res) => res.send('Get all users'));
router.get('/:id', (req, res) => res.send('Get user by id'));
router.post('/', (req, res) => res.send('Create user'));

module.exports = router;
```

This keeps your code clean, scalable and easy to maintain.

6 Complete Routing Example

```
const express = require('express');
const app = express();
app.use(express.json()); // to parse JSON body

// GET all users
app.get('/users', (req, res) => {
  res.json({ message: 'Get all users' });
});
// GET user by id
app.get('/users/:id', (req, res) => {
  res.json({ userId: req.params.id });
});
// POST create user
app.post('/users', (req, res) => {
  const { name, email } = req.body;
  res.status(201).json({ message: 'User created', name, email });
});
app.listen(3000, () => console.log('Server running on http://localhost:3000'));
```

Tip: Use `app.use(express.json())` to parse incoming JSON data.

7 Route Matching Order

Express matches routes in the order they are defined.

1. Specific routes first — e.g. /users/profile
2. Then route with params — e.g. /users/:id
3. Then general routes — e.g. /users
4. Then catch-all routes — e.g. * (Not Found)

Example Order

```
app.get('/users/profile', handler1);
app.get('/users/:id', handler2);
app.get('/users', handler3);
app.all('*', handlerNotFound);
```

Wrong order can cause specific routes not to work as expected!

Common Built-in Properties

Property	Description
req.params	Route parameters (e.g. /users/:id)
req.query	Query parameters (e.g. ?q=node)
req.body	Data sent in request body (POST/PUT/PATCH)
req.method	HTTP method (GET, POST, PUT, DELETE, etc.)
req.path	Request path (e.g. /users/1)

Quick Reference

- GET — Read data
- POST — Create data
- PUT — Replace data
- PATCH — Partial update
- DELETE — Remove data

Example URLs

- GET /users
- GET /users/123
- GET /search?q=node&page=1
- POST /users
- PUT /users/123
- PATCH /users/123
- DELETE /users/123

Best Practices

- ✓ Keep routes RESTful and consistent.
- ✓ Use meaningful route names.
- ✓ Validate and sanitize user input.
- ✓ Use Router for modular code.
- ✓ Handle errors properly.
- ✓ Use proper status codes.

Remember

GET -> Retrieve data
POST -> Create new data
PUT -> Update entire data
PATCH -> Update partial data
DELETE -> Delete data

Express.js Middleware

Functions that run between the Request and Response

Modular — Organize code in reusable components

Reusable — Use multiple times across routes

Control Flow — Use *next()* to pass control to the next middleware

Secure — Add authentication, validation and error handling

Powerful — Extend Express functionality easily

Middleware

Middleware are functions that have access to the request object (*req*), response object (*res*) and the *next* middleware function in the application's request-response cycle.

app.use()

The *app.use()* function is used to register middleware in your Express application. It can be used for all routes or specific paths.

next()

The *next()* function passes control to the next middleware in the stack. If not called, the request will hang.

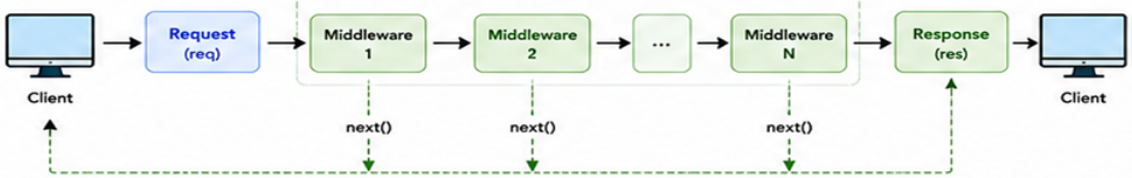
Request (req)

The *req* object contains information about the HTTP request such as headers, params, body, query and more.

Response (res)

The *res* object is used to send responses back to the client like JSON, views, files, redirects, status codes, etc.

Middleware Flow



Basic Middleware Example

```
1  const express = require('express');
2  const app = express();
3
4  // Middleware function
5  const logger = (req, res, next) => {
6    console.log('${req.method} ${req.url}');
7    next(); // Pass control to the next middleware
8  };
9  app.get('/', logger, (req, res) => {
9    res.send('Hello Middleware!');
10 });
```

What's Happening?

- *logger* runs first
- It logs the method and URL
- *next()* calls the next function (route handler)
- Finally, the response is sent to the client

Run: `node app.js`

1 Logger Middleware

```
app.use((req, res, next) => {
  console.log('[${new Date().toISOString()}]',
              req.method, req.url);
  next();
});
```

Logs every incoming request.

2 Authentication Middleware

```
app.use((req, res, next) => {
  const token = req.headers.authorication;
  if (!token) {
    return res.status(401).json({ error: 'Unauthorized' });
  }
  next();
});
```

Checks authentication before allowing access.

3 Error Handler Middleware

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({
    error: 'Something went wrong!'
  });
});
```

Handles errors and sends a proper response.

Order Matters

1. Logger Middleware
2. Authentication Middleware
3. Routes
4. Error Handler

Middleware are executed in the order they are registered.

Place error handler at the end of middleware stack.

Common Built-in Middleware

Middleware	Description	Usage
express.json()	Parse JSON request bodies	app.use(express.json())
express.urlonceded()	Parse URL-encoded bodies	app.use(express.urlencoded({ extended: true }))
express.static()	Serve static files	app.use(express.static('public'))
cookie-parser	Parse cookies	app.use(require('cookie-parser')())
morgan	HTTP request logger	app.use(require('morgan')('dev'))

Quick Summary

- Middleware functions have access to *req*, *res* and *next*.
- Use *next()* to pass control.
- Use *app.use()* to register middleware.
- Order of middleware is important.
- Handle errors with error-handling middleware (4 agruments).

Pro Tip

Keep middleware small, focused and reusable. This makes your code clean, maintainable and easy to debug!

Middleware = The Secret Sauce of Express Apps!

Official Docs: https://expressjs.com/en/guide/writing-middleware.html

Master Middleware. Build Better APIs.

REST API

– Simple • Stateless • Scalable

REST (Representational State Transfer) is an architectural style for building web services using **HTTP**.

Stateless
No client context stored on server

Resource Based
Everything is a resource (URI)

Standard Methods
Uses HTTP methods for operations

Cacheable
Responses can be cached

Layered System
Supports multiple layers

Endpoints (Examples)

Method	Endpoint	Description
GET	/api/users	Get all users
GET	/api/users/1	Get user by ID
POST	/api/users	Create a new user
PUT	/api/users/1	Update entire user
PATCH	/api/users/1	Update partial user
DELETE	/api/users/1	Delete user

JSON (Data Format)

```
{
  "id": 1,
  "name": "Lovejeet Singh",
  "email": "lovejeet@example.com",
  "role": "Developer",
  "active": true,
  "createdAt": "2025-06-01T10:30:00Z"
}
```

JSON is the most common format for sending and receiving data in REST APIs.

HTTP Methods

GET	Retrieve data	Read a resource
POST	Create data	Create a new resource
PUT	Update data (replace)	Replace entire resource
PATCH	Update data (partial)	Update partial resource
DELETE	Delete data	Remove a resource

Example: Get User

Request

```
GET /api/users/1 HTTP/1.1
Host: api.example.com
Accept: application/json
Authorization: Bearer token123
```

Response (200 OK)

```
{
  "id": 1,
  "name": "Lovejeet Singh",
  "email": "lovejeet@example.com",
  "role": "Developer"
}
```

Best Practices

- Use nouns for resources
- Use proper HTTP methods
- Return meaningful status codes
- Secure your APIs
- Version your API

Status Codes

Code Range	Code	Meaning	Example
1xx	100	Continue	Request received, continue
2xx	200	OK	Request successful
	201	Created	Resource created successfully
	204	No Content	Request successful, no content
3xx	301	Moved Permanently	Resource moved permanently
	304	Not Modified	Resource not modified
4xx	400	Bad Request	Invalid request
	401	Unauthorized	Authentication required
	404	Not Found	Resource not found
	405	Method Not Allowed	HTTP method not allowed
5xx	500	Internal Server Error	Server error
	503	Service Unavailable	Server temporarily unavailable

Quick Reference

- **Base URL:** https://api.example.com
- **Data Format:** JSON
- **Authentication:** Token / OAuth / API Key
- All responses are stateless
- Use HTTPS for all requests

Tip: Keep your API simple, consistent and well documented.

REST API Power your apps. Connect the world. | JSON over HTTP | Resources with URIs | Standard Methods | Secure & Scalable

CRUD OPERATIONS

Create • Read • Update • Delete

The 4 basic operations for managing data in web applications

SIMPLE — Easy to understand and implement

STANDARD — Industry standard operations

RELIABLE — Consistent and predictable

SCALABLE — Works for small to large applications

CREATE

`POST` `/api/users`

Create a new resource

Request Body (JSON)

```
{
 "name": "Lovejeet Singh",
 "email": "lovejeet@example.com",
 "role": "Developer"
}
```

Response (201 Created)

```
{
  "id": 1,
  "name": "Lovejeet Singh",
  "email": "lovejeet@example.com",
  "role": "Developer",
  "createdAt": "2025-06-01T10:30:00Z"
}
```

READ

`GET` `/api/users/{id}`

Retrieve a resource by ID or list all resources

Response (200 OK)

```
{
  "id": 1,
  "name": "Lovejeet Singh",
  "email": "lovejeet@example.com",
  "role": "Developer",
  "createdAt": "2025-06-01T10:30:00Z",
  "updatedAt": "2025-06-01T12:00:00Z"
}
```

UPDATE

`PUT` `/api/users/{id}`

Update an existing resource (replace entire resource)

Request Body (JSON)

```
{
  "name": "Lovejeet Singh",
  "email": "lovejeet@example.com",
  "role": "Senior Developer"
}
```

Response (200 OK)

```
{
  "id": 1,
  "name": "Lovejeet Singh",
  "email": "lovejeet@example.com",
  "role": "Senior Developer",
  "updatedAt": "2025-06-01T12:30:00Z"
}
```

DELETE

`DELETE` `/api/users/{id}`

Delete a resource

Response (200 OK)

```
{
  "message": "User deleted successfully",
  "id": 1
}
```

DATABASE

CRUD operations interact with the database to manage your data.

- Stores all application data
- Ensures data integrity
- Supports relationships
- Handles transactions
- Provides backup & recovery

REQUEST & RESPONSE

Request

- HTTP Method
- Endpoint (URL)
- Headers
- Request Body (JSON)
- Query Parameters
- Authentication

Response

- Status Code
- Response Body (JSON)
- Headers
- Error Message (if any)
- Pagination (if applicable)

VALIDATION

Always validate input data before processing.

- Required fields
- Data types
- Format (Email, Phone, etc.)
- Length & Range
- Unique values
- Business rules

Example Validation (User)

```
{
   "name": "required|string|max:100",
   "email": "required|email|unique:users,email",
   "role": "required|in:Developer,Manager,Admin"
}
```

Good validation = Secure & Reliable API

DXAMPLE: USER RESOURCE ENDPOINTS

Method	Endpoint	Description	Example
POST	/api/users	Create a new user	Create user
GET	/api/users	Get all users	Get all users
GET	/api/users/1	Get user by ID	Get user with ID 1
PUT	/api/users/1	Update entire user	Update user with ID 1
PATCH	/api/users/1	Update partial user	Update user name
DELETE	/api/users/1	Delete user	Delete user with ID 1

HTTP STATUS CODES

Code	Meaning	When Used
200	OK	Request successful
201	Created	Resource created successfully
204	No Content	Request successful, no content to return
400	Bad Request	Invalid request or data
401	Unauthorized	Authentication required or failed
404	Not Found	Resource not found
500	Internal Server Error	Server-side error

BEST PRACTICES

- Use proper HTTP methods for each operation
- Validate all input data
- Return meaningful status codes
- Use consistent response format
- Implement authentication & authorization
- Handle errors gracefully
- Document your API endpoints
- Use pagination for large datasets

Consistency is Key!

Follow the same patterns across your API for better maintainability and developer experience.

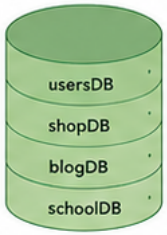
MongoDB BASICS

NoSQL Database for Modern Applications

NoSQL — Schema-less, Flexible
Scalable — High Performance, Horizontal Scaling
Reliable — High Availability, Data Redundancy
Flexible — JSON-like Documents
Open Source — Community Driven

DATABASE

A container for collections.
It holds your data.



COLLECTION

A group of MongoDB documents.
Similar to a table in RDBMS.

users (Collection)
- Document 1
- Document 2
- Document 3
- ...

DOCUMENT

A record in a collection.
Stored in BSON (JSON-like).

```
{
  "_id": ObjectId("665c5fbc8e1e9a1f8b456789"),
  "name": "Lovejeet Singh",
  "email": "lovejeet@example.com",
  "age": 28,
  "role": "Developer",`,
  "skills": ["Node.js", "MongoDB", "Express"],
  "createdAt": ISODate("2025-06-01T10:30:00Z")
}
```

FIELD

Documents are made up of key-value pairs.

_id	→	ObjectId
name	→	String
email	→	String
age	→	Number
role	→	String
skills	→	Array
createdAt	→	Date

MONGODB COMPASS

The official GUI for MongoDB.

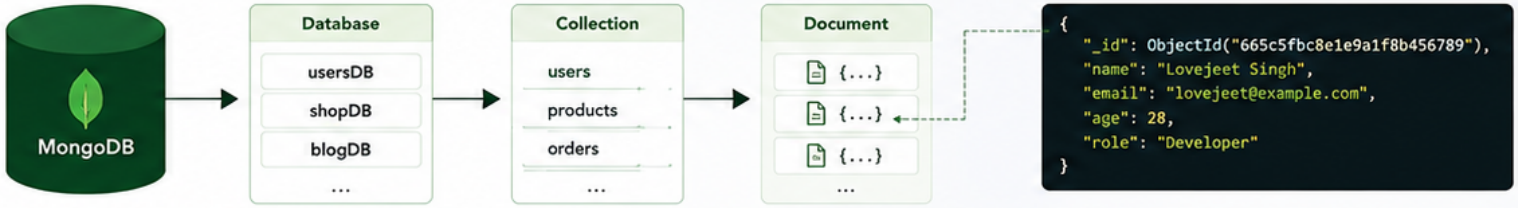
- ✓ Visualize your data
- ✓ Create, read, update, delete data
- ✓ Analyze performance
- ✓ Build queries easily

MONGO SHELL

The command-line interface for MongoDB.

- ✓ Powerful querying
- ✓ Scripting & automation
- ✓ Database administration
- ✓ Available in `mongo shell` or `mongosh`

HOW DATA IS ORGANIZED IN MONGODB



KEY CONCEPTS

NoSQL	MongoDB is a NoSQL database that stores data in flexible, JSON-like documents.
Schema-less	No fixed schema. You can insert different fields in different documents.
BSON	Binary JSON format used to store documents internally.
High Performance	Built for speed, scale, and developer productivity.

BASIC COMMANDS (MONGO SHELL)

```
// Show all databases
show dbs

// Use a database
use usersDB

// Show collections
show collections
```

```
// Insert a document
db.users.insertOne({ name: "John", age: 25 })

// Find all documents
db.users.find()

// Find with condition
db.users.find({ age: { $gt: 20 } })
```

```
// Update a document
db.users.updateOne({ name: "John" }, { $set: { age: 26 } })

// Delete a document
db.users.deleteOne({ name: "John" })

// Count documents
db.users.countDocuments()
```

_id / OBJECTID

Every document has a unique _id field of type ObjectId.

```
ObjectId("665c5fbc8e1e9a1f8b456789")
```

- ✓ 12 bytes / 24 hex characters
- ✓ Generated automatically if not provided
- ✓ Ensures uniqueness

QUICK SUMMARY

MongoDB	A NoSQL document database
Database	Container for collections
Collection	Group of documents
Document	JSON-like data record
Compass	GUI tool for MongoDB
Mongo Shell	CLI tool for MongoDB
ObjectId	Unique ID for every document

MongoDB — The Developer Data Platform
Store Anything — From any app, anywhere
Query Anything — Powerful and flexible
Scale Anywhere — On-prem or in the cloud
Build Faster — Focus on your app, not schema

Mongoose

Elegant MongoDB object modeling for Node.js

Simplify your MongoDB interactions with Mongoose ODM

ODM for MongoDB — Model your data in a structured way

Easy & Powerful — Less boilerplate, more productivity

Built-in Validation — Validate data before saving to database

Middleware — Pre & post hooks for queries & docs

Performant — Built for speed and scalability

Connection

Establish connection with MongoDB.

```
mongoose.connect(uri, options)
```

Schema

Define the shape and structure of documents.

```
new mongoose.Schema({ ... })
```

Model

A compiled Schema. Interface to the database.

```
mongoose.model(name, schema)
```

Document

Instance of a Model. Represents a single record.

```
new Model(data)
```

Validation

Built-in and custom validators for data.

```
required, min, enum, match, etc.
```

Populate

Replace references with actual documents.

```
.populate('field')
```

CRUD

Create, Read, Update, Delete operations.

```
save(), find(), update(), etc.
```

1. Connection

```
const mongoose = require('mongoose');

mongoose.connect('mongodb://127.0.0.1:27017/myDB', {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
  .then(() => console.log('✔ Connected to MongoDB'))
  .catch(err => console.error('✖ Connection Error:', err));
```

Always handle connection errors in production.

2. Schema, Model & Document Flow

Schema

name	String
email	String
age	Number
...	

Define structure and rules for data.

Model

User (Model)

Compile Schema into a Model (constructor).

Document

```
{
  name: "Lovejeet",
  email: "love@ex.com",
  age: 28
}
```

Create instances (docs) from Model.

3. Example: User Schema, Model & Document

```
const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  age:  { type: Number, min: 0 },
  role: { type: String, enum: ['User', 'Admin'], default: 'User' }
}, { timestamps: true });

const User = mongoose.model('User', userSchema);

const user = new User({
  name: 'Lovejeet Singh',
  email: 'love@example.com',
  age: 28,
  role: 'Developer'
});
await user.save();
  console.log('User saved!');
```

Saved Document (in MongoDB)

```
{
  _id:  ObjectId("665c5fbc8e1e9a1f8b456789"),
  name:   "Lovejeet Singh",
  email:  "love@example.com",
  age:  28,
  role:   "Developer",
  createdAt:  ISODate("2025-06-01T10:30:00Z"),
  updatedAt:  ISODate("2025-06-01T10:30:00Z"),
  __v:  0
}
```

`_id` is of type `ObjectId` and is created automatically.

4. Validation

```
const userSchema = new mongoose.Schema({
  email: { type: String, required: true, unique: true },
  age: { type: Number, min: [0, 'Age must be positive'] },
  name: { type: String, trim: true, minlength: 2 }
});
```

Custom Validator Example

```
userSchema.path('age').validate(v => v < 120,
  'Age must be less than 120');
```

5. Populate (References)

Post Schema

```
const postSchema = new mongoose.Schema({
  title: String,
  content: String,
  author: { type: mongoose.Schema.Types.ObjectId,
        ref: 'User' }
});
```

Query with Populate

```
const posts = await Post.find()
  .populate('author', 'name email')
  .exec();
// author field will contain full
// User document (name, email)
```

ObjectId (ref) → Populated Document

6. CRUD Operations

Create

```
const user = new User({ name: 'John', email: 'john@ex.com' });
await user.save();
```

Read

```
const users = await User.find({ age: { $gte: 18 } });
const user = await User.findById(id);
```

Update

```
await User.updateOne({ _id: id }, { $set: { age: 29 } });
await User.findByIdAndUpdate(id, { name: 'John Doe' }, { new: true });
```

Delete

```
await User.deleteOne({ _id: id });
await User.findByIdAndDelete(id);
```

Why Mongoose?

- ✓ Simple and intuitive API
- ✓ Schema validation
- ✓ Middleware & Hooks
- ✓ Population of references
- ✓ Better developer experience
- ✓ Production ready

Schema First — Design your data structure

Validate Data — Keep your data clean

Use References — Normalize relationships

Index for Performance — Add indexes to speed up queries

Handle Errors — Always use try/catch

Mongoose — Better with MongoDB

Environment Variables

Store sensitive & configuration data outside your code

Secure | Flexible | Maintainable | 12-Factor App

.env — Environment file stores key=value pairs

dotenv — Loads .env variables into process.env

process.env — Access env vars in your Node.js application

Keep Secrets — Never commit .env files to version control

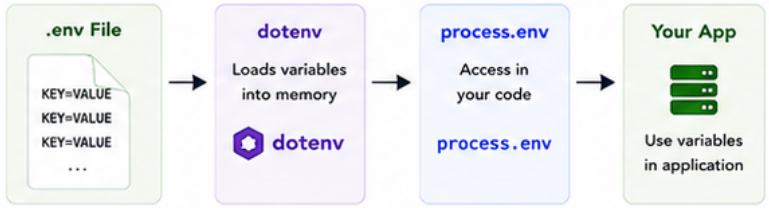
What is .env?

A .env file stores environment variables as KEY=VALUE pairs.

```
PORT=5000
MONGO_URI=mongodb://localhost:27017/myapp
JWT_SECRET=supersecretjwtkey
API_KEY=sk_live_1234567890abcdef
NODE_ENV=development
FRONTEND_URL=http://localhost:3000
EMAIL_USER=yourmail@gmail.com
EMAIL_PASS=yourpassword
```

Never hardcode secrets in your code! Use environment variables.

How it Works



Tip: Add .env to .gitignore to keep secrets safe.

Accessing in Code

```
1  require('dotenv').config();
2
3  const port = process.env.PORT;
4  const mongoURI = process.env.MONGO_URI;
5  const jwtSecret = process.env.JWT_SECRET;
6  const apiKey = process.env.API_KEY;
7
8  console.log('Server running on port', port);
9  console.log('Mongo URI:", mongoURI);
10
11 app.listen(port, () => {
12    console.log(`Server is running on port ${port}`);
13 });
```

Must call dotenv.config() at the very top of your entry file.

Common Environment Variables

Variable	Purpose	Example
PORT	Port number for the server	PORT=5000
MONGO_URI	MongoDB connection string	MONGO_URI=mongodb://localhost:27017/myapp
JWT_SECRET	Secret key for JWT	JWT_SECRET=supersecretjwtkey
API_KEY	Third-party API key	API_KEY=sk_live_1234567890abcdef
NODE_ENV	Environment (development/production)	NODE_ENV=development
FRONTEND_URL	Frontend application URL	FRONTEND_URL=http://localhost:3000
EMAIL_USER	Email service username	EMAIL_USER=yourmail@gmail.com
EMAIL_PASS	Email service password	EMAIL_PASS=yourpassword

Example Project Structure

```
my-app/
|-- .env
|-- .gitignore
|-- package.json
|-- server.js
`-- config/
    `-- db.js
```

Add .env to .gitignore

```
# .gitignore
.env
node_modules/
logs/
.DS_Store
```

This keeps your secrets safe and out of GitHub.

Best Practices

- Never commit .env files.
- Use different .env files for different environments.
- Use strong secrets and rotate them regularly.
- Validate required env variables at startup.
- Use a config file to centralize access.

If a variable is missing, fail fast and show ta clear error.

Example .env Files for Environments

.env.development

```
PORT=5000
MONGO_URI=mongodb://localhost:27017/myapp
NODE_ENV=development
```

.env.production

```
PORT=8000
MONGO_URI=mongodb+srv://user:pass@cluster/db
NODE_ENV=production
```

.env.test

```
PORT=5001
MONGO_URI=mongodb://localhost:27017/testdb
NODE_ENV=test
```

Quick Summary

AUTHENTICATION

Secure Users. Protect Data. Build Trust.

Register • Login • Password • Hashing • Security • User

USER — Unique identity for every user

SECURITY — Keep data safe and protected

AUTHENTICATION — Verify identity before access

AUTHORIZATION — Allow access to authorized users

DATA PROTECTION — Hash passwords & secure tokens

1 REGISTER

Create a new user account

- Enter name, email and password
- Validate input (email unique, etc.)
- Hash password using bcrypt
- Save user in database

Example Request (JSON)

```
{
  "name": "Lovejeet Singh",
  "email": "lovejeet@example.com",
  "password": "mypassword123"
}
```

Response (201 Created)

```
{
  "message": "User registered
    successfully"
}
```

LOGIN

Authenticate user & generate token

- Enter email and password
- Find user by email
- Compare password with hashed password
- If valid, generate JWT token
- Send token in response

Example Request (JSON)

```
{
  "email": "lovejeet@example.com",
  "password": "mypassword123"
}
```

Response (200 OK)

```
{
  "message": "Login successful",
  "token": "eyJhbGciOiJIUzI1Niis..."
}
```

PASSWORD HASHING WITH BCRYPT

Never store plain text passwords!

`mypassword123` (Plain Password) → bcrypt → `$2b$10$KIXQmZ4KQZQp8Q8zQe8uFuO7kJ...` (Hashed Password)

```
// Hash Password
const bcrypt = require('bcrypt');
const salt = await bcrypt.genSalt(10);
const hashedPassword = await bcrypt.hash('mypassword123', salt);
// Compare Password
const isMatch = await bcrypt.compare('mypassword123', hashedPassword);
```

USER MODEL EXAMPLE (MONGOOSE)

```
const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, unique: true, required: true },
  password: { type: String, required: true },
  createdAt: { type: Date, default: Date.now }
});
module.exports = mongoose.model('User', userSchema);
```

Always validate and sanitize user input.

AUTHENTICATION FLOW

USER — User sends credentials → **SERVER** — Validate user credentials → **DATABASE** — Check user & hashed password → **TOKEN** — If valid, generate JWT → **ACCESS** — User access protected routes

JWT Token is sent in Authorization header for protected routes.
Example: `Authorization: Bearer <token>`

WHY HASH PASSWORDS?

- Protects passwords if database is leaked
- One-way encryption (cannot revert back)
- Each hash is unique (uses salt)
- Industry standard for security

`mypassword123` (Plain Text) → `$2b$10$KIXQmZ4KQZQp...` (Hashed (Safe))

SECURITY BEST PRACTICES

- Use bcrypt to hash passwords
- Validate email and inputs
- Use JWT for authentication
- Set token expiration
- Never store plain passwords
- Use HTTPS in production
- Keep secrets in .env file

QUICK SUMMARY

Register	Create user, hash password, save in DB
Login	Verify credentials, return JWT token
Password	Hash with bcrypt, never store plain text
Authentication	Use token to access protected routes
Security	Validation, hashing, HTTPS, .env secrets

Authentication — Is the first step to security

Protect Users — Secure accounts & data

Secure Code — Better practices, safer apps

Build Trust — Security builds confidence

Security First — Always & Everywhere

JWT AUTHENTICATION

Secure. Stateless. Scalable.

JWT | Token | Bearer | Secure | Stateless

1. **LOGIN** — User logs in with credentials
2. **SIGN** — Server signs a JWT using Secret Key
3. **TOKEN** — JWT is sent to client
4. **SEND** — Client sends token in Authorization header
5. **VERIFY** — Server verifies token
6. **ACCESS** — Access to protected route

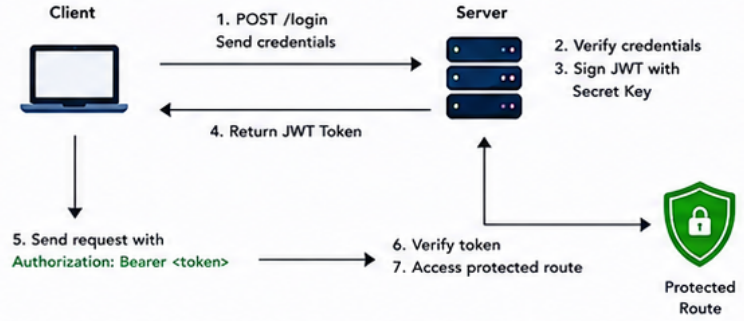
WHAT IS JWT?

JWT (JSON Web Token) is a compact, URL-safe token used to securely transmit information between parties as a JSON object. It is commonly used for Authentication.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJ1c2VySWOiOjEsImVtYWlsIjoibG92ZWpjZXRA
ZXhhbXBsZ5jb20iLCJpYXQiOjE3MTYwMDAwMDB9 .
X5Qv...Zr3JQ
```

- **HEADER** — ALGORITHM & TOKEN TYPE
- **PAYLOAD** — DATA (CLAIMS)
- **SIGNATURE** — VERIFY INTEGRITY

HOW JWT WORKS



JWT STRUCTURE

Header

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Contains the type of token and signing algorithm.

Payload

```
{
  "userId": 1,
  "email": "lovejeet@example.com",
  "role": "admin",
  "iat": 1716000000,
  "exp": 1716086400
}
```

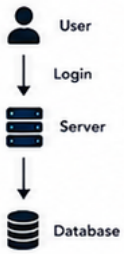
Contains the claims (data) about the user and token metadata.

Signature

```
HMACSHA256(
base64UrlEncode(header) + "." +
base64UrlEncode(payload), secretKey
)
```

Ensures the token is not changed. Created using Secret Key.

AUTHENTICATION FLOW (EXAMPLE)



1. User sends email & password
2. Server verifies user in database
3. Server creates payload (claims)
4. Server signs JWT using Secret Key
5. JWT is sent to user (client)
6. Client stores token (localStorage / cookie)
7. Client sends request with token in header
8. Server verifies token
9. If valid → Allow access to protected route
10. If invalid/expired → Deny access

EXAMPLE: GENERATE & VERIFY JWT (NODE.JS)

SIGN (GENERATE TOKEN)

```
const jwt = require('jsonwebtoken');
const secretKey = process.env.JWT_SECRET; // secret key

const payload = {
  userId: 1,
  email: 'lovejeet@example.com',
  role: 'admin'
};

const token = jwt.sign(payload, secretKey, { expiresIn: '1h' });
console.log(token);
```

VERIFY TOKEN

```
const jwt = require('jsonwebtoken');
const secretKey = process.env.JWT_SECRET;

function verifyToken(token) {
  try {
    const decoded = jwt.verify(token, secretKey);
    return decoded; // valid
  } catch (err) {
    throw new Error('Invalid or Expired Token');
  }
}
```

AUTHORIZATION HEADER

Client must send the token in each request header.

```
GET /api/protected HTTP/1.1
Host: example.com
Authorization: Bearer <your_jwt_token>
Content-Type: application/json
```

Always use HTTPS in production to keep tokens safe.

COMMON CLAIMS (PAYLOAD)

iss	Issuer
sub	Subject (User ID)
aud	Audience
exp	Expiration Time
iat	Issued At
nbf	Not Before
jti	JWT ID (unique)

```
{
  "iss": "your-app",
  "sub": "1",
  "email": "lovejeet@example.com",
  "role": "admin",
  "iat": 1716000000,
  "exp": 1716086400
}
```

TOKEN TYPE

JWT is a compact string token used for securely transmitting information.

STATELESS

Server doesn't store session. All info is inside the token (claims).

BEARER TOKEN

Token is used as a Bearer token in Authorization header.

SECRET KEY

Keep your JWT Secret Key strong and never expose it in client-side code.

EXPIRATION

Always set an expiration time for tokens to improve security.

BEST PRACTICES

- Use strong secret keys
- Keep tokens short-lived
- Store token securely
- Use HTTPS always
- Implement refresh tokens
- Validate & verify tokens on every request
- Don't store sensitive data in payload

COOKIES

HTTP State Management

Store small amounts of data on the client.

cookie-parser | Secure | HTTP Only | SameSite | Session

HOW COOKIES WORK

WHAT IS A COOKIE?

Cookies are small text files stored in the user's browser by a website. They help remember information like preferences, login status, and session data.

INSTALL & USE COOKIE-PARSER

```
# Install
npm install cookie-parser

# Use in Express app
const express = require('express');
const cookieParser = require('cookie-parser');

const app = express();
app.use(cookieParser());

// Now you can use req.cookies and res.cookie()
// in your routes
```

SET COOKIE

Send a cookie to the browser.

```
// Set cookie
res.cookie('username', 'lovejeet', {
  maxAge: 60 * 60 * 1000, // 1 hour
  httpOnly: true,
  secure: true,    // use in HTTPS
  sameSite: 'Lax'
});

// Simple cookie
res.cookie('theme', 'dark');
```

READ COOKIE

Read cookies from the request.

```
// Read a cookie
const username = req.cookies.username;
console.log(username);

// Read all cookies
console.log(req.cookies);

// Check if cookie exists
if (req.cookies.token) {
  console.log('Token exists');
}
```

COOKIE ATTRIBUTES

HTTP Only	`httpOnly: true`	Prevents JavaScript access to the cookie. Helps protect against XSS attacks.
Secure	`secure: true`	Cookie is only sent over HTTPS connections.
SameSite	`sameSite: 'Lax'`	Controls when cookies are sent with cross-site requests. Options: Strict | Lax | None
Expires / Max-Age	`maxAge: 3600000`	Defines cookie lifetime. Use maxAge (in ms) or expires.
Path	`path: '/'`	Specifies the URL path for which the cookie is valid.
Domain	`domain: 'example.com'`	Specifies the domain for which the cookie is valid.

SESSION WITH COOKIES

Sessions store data on the server and use a cookie (session ID) on the client.

- Server stores session data.
- Browser stores only the session ID in cookie.
- More secure and scalable.

COOKIE EXAMPLE (LOGIN FLOW)

1. User submits login form
2. Server validates credentials
3. Server sets a token/session cookie
4. Browser stores the cookie
5. Browser sends cookie on every request
6. Server verifies cookie and allows access

DELETE COOKIE

Remove a cookie.

```
res.clearCookie('username');
// or with options (must match set options)
res.clearCookie('username', { path: '/' });
```

CHECK COOKIE

Check if a cookie exists.

```
if (req.cookies.username) {
  console.log('Cookie exists');
} else {
  console.log('Cookie not found');
}
```

ALL COOKIES

Get all cookies.

```
console.log(req.cookies);
/* Output Example:
{
  username: 'lovejeet',
  theme:  'dark'
}
*/
```

COOKIE vs SESSION

Feature	Cookie	Session
Storage	Client (Browser)	Server
Size	~4KB	Unlimited
Security	Less (unless secured)	More Secure
Expires	Set by client/server	Server controlled
Use Case	Preferences, Tokens	User login, Cart, Data
Data Stored	Small amount	Large amount

BEST PRACTICES

- Always use HTTP Only and Secure cookies.
- Use SameSite=Lax or Strict to prevent CSRF.
- Store only necessary data in cookies.
- Avoid storing sensitive data in cookies.
- Use short expiration time for cookies.
- Use sessions for storing larger and sensitive data.

Cookies — Small but powerful | **cookie-parser** — Middleware for Express | **Secure** — Protect user data | **HTTP Only** — Prevent XSS attacks | **SameSite** — Prevent CSRF attacks | **Session** — Better for sensitive data

SESSIONS

Manage user state on the server.

express-session | Secure | Server Side | Authentication | Persistent

HOW SESSIONS WORK

1. **Client Request** — Browser sends request to server
2. **Server Creates Session** — Server generates Session ID (SID)
3. **Set-Cookie (SID)** — SID is sent to client in a cookie
4. **Subsequent Requests** — Client sends SID cookie with every request
5. **Retrieve Session** — Server retrieves session data using SID

WHAT IS A SESSION?

A session stores data on the server and links it to a user using a unique Session ID (SID) stored in a cookie on the client.

1 INSTALL & SETUP express-session

```
# Install
npm install express-session

# Basic Setup in Express
const session = require('express-session');

app.use(session({
  secret: 'lovejeet_secret_key',  // change this!
  resave: false,
  saveUninitialized: false,
  cookie: { maxAge: 1000 * 60 * 60, httpOnly: true }
}));
```

Always use a strong secret in production and store it in environment variables.

2 SESSION FLOW (LOGIN)

User submits login form → Server validates credentials → Create session and store data → Send SID cookie to browser → Browser sends SID with every request → Server retrieves session data

```
// Store user in session after login
req.session.user = {
  id: user.id,
  name: user.name,
  email: user.email
};

req.session.isAuthenticated = true;
```

What is stored?

- ✓ User ID / Info
- ✓ Authentication status
- ✓ Cart items
- ✓ Preferences
- ✓ Any temporary data

3 LOGOUT (DESTROY SESSION)

```
// Logout user
req.session.destroy(err => {
  if (err) return next(err);
  res.clearCookie('connect.sid'); // clear cookie
  res.redirect('/login');
});
```

Always clear the session and cookie on logout!

SESSION ID (SID)

A unique ID is generated for each session.
Stored in a cookie named connect.sid by default.

Example Cookie

```
connect.sid=s%3AAbc123Xyz...;
HttpOnly; Path=/;
```

- ✓ This ID is just a key.
- ✓ Actual data is stored on the server.

SESSION OPTIONS (COOKIE)

secret	Secret key to sign the session ID cookie
resave	Force session to be saved back to store
saveUninitialized	Save uninitialized sessions
cookie.maxAge	Session expiration time (in ms)
cookie.httpOnly	Prevent client-side JS access (XSS protection)
cookie.secure	Send cookie only over HTTPS
cookie.sameSite	Control CSRF (strict / lax / none)

4 SESSION STORES

MEMORY STORE (Default)

- ✓ Stores session in server memory.
- ✓ Fast but not suitable for production.
- ✓ Data is lost when server restarts.

```
const session = require('express-session');
const MemoryStore = require('memorystore')(session);

app.use(session({
  store: new MemoryStore({ checkPeriod: 86400000 }),
  secret: 'lovejeet_secret_key',
  resave: false,
  saveUninitialized: false
}));
```

⚠ Do NOT use MemoryStore in production!

REDIS STORE (Recommended)

- ✓ Stores session in Redis.
- ✓ Persistent, scalable & production ready.
- ✓ Session data survives server restarts.

```
const session = require('express-session');
const RedisStore = require('connect-redis')(session);
const { createClient } = require('redis');

const client = createClient({ url: 'redis://127.0.0.1:6379' });
client.connect().then(() => console.log('Redis Connected'));

app.use(session({
  store: new RedisStore({ client }),
  secret: 'lovejeet_secret_key',
  resave: false,
  saveUninitialized: false,
  cookie:initialized: 1000 * 60 * 60, httpOnly: true }
}));
```

DATABASE STORE (Other Options)

- ✓ MongoDB, MySQL, etc.
- ✓ Good for long-term session storage.
- ✓ Use connect-mongo, connect-session-knex.

Popular Options

connect-mongo	MongoDB
connect-session-knex	SQL DB
connect-session-sequelize	Sequelize

5 AUTHENTICATION WITH SESSIONS

1. **User Login** — Enters credentials
2. **Authenticate** — Validate user
3. **Create Session** — Store user info in session
4. **Access Protected Routes** — Check session & allow access

```
// Middleware to protect routes
function isAuthenticated(req, res, next) {
  if (req.session.isAuthenticated) {
    return next();
  }
  res.redirect('/login');
}
app.get('/dashboard', isAuthenticated,
  (req, res) => {
    res.send('Welcome to Dashboard!');
} )
```

Use middleware to protect private routes.

BEST PRACTICES

- ✓ Use HTTPS in production
- ✓ Set cookie.httpOnly = true
- ✓ Set cookie.secure = true (in production)
- ✓ Use strong secret and keep it safe
- ✓ Use Redis or DB store in production
- ✓ Regenerate session ID after login
- ✓ Destroy session on logout
- ✓ Set appropriate session expiration

QUICK SUMMARY

Feature	Description
express-session	Middleware to manage sessions in Express
Session ID (SID)	Unique ID stored in cookie (connect.sid)
Store	Where session data is saved on server
Memory Store	Default store (not for production)
Redis Store	Best for production (fast & persistent)

SESSION DATA EXAMPLE

```
req.session.user = {
  id: 1,
  name: 'Lovejeet Singh',
  email: 'lovejeet@example.com',
  role: 'admin'
};

// Access later
console.log(req.session.user.name);
```

KEY TAKEAWAYS

- ✓ Sessions store data on the server.
- ✓ Client only holds the Session ID.
- ✓ Use secure cookies and proper store.
- ✓ Sessions are great for authentication.
- ✓ Always destroy session on logout.

Sessions = State + Security + Simplicity

FILE UPLOAD with MULTER

Easily upload files in your Node.js / Express app

Multer | Upload | Storage | File Filter | Secure

HOW FILE UPLOAD WORKS

1. **Client** — Selects file(s) and submits the form
2. **Express Server** — Receives the request with multipart/form-data
3. **Multer** — Processes the file(s) and saves to storage
4. **Storage** — File is saved in destination
5. **Response** — Server sends success response

WHAT IS MULTER?

Multer is a Node.js middleware for handling multipart/form-data, which is primarily used for uploading files. It is written on top of busboy for maximum efficiency.

MULTER

INSTALL MULTER

Install using npm

```
npm install multer
```

BASIC SETUP

```
const express = require('express');
const multer = require('multer');
const app = express();

app.use(express.json());
const upload = multer({ dest: 'uploads/' });

// Your routes here
app.listen(3000, () => {
  console.log('Server running on port 3000'));
```

Don't forget to create 'uploads' folder or Multer will create it automatically.

STORAGE ENGINES

DISK STORAGE (Default)

Stores files on disk.

```
const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, 'uploads/');
  },
  filename: function (req, file, cb) {
    const unique = Date.now() + '-' + file.originalname;
    cb(null, unique);
  }
});
```

Files are saved permanently on server disk.

MEMORY STORAGE

Stores files in memory as Buffer.

```
const storage = multer.memoryStorage();
const upload = multer({ storage: storage });
```

Useful for processing files before saving (e.g., upload to cloud like S3, Cloudinary).

FILE FILTER

Filter files by extension, mimetype or custom logic.

```
const fileFilter = (req, file, cb) => {
  const allowed = ['image/jpeg', 'image/png', 'application/pdf'];
  if (allowed.includes(file.mimetype)) {
    cb(null, true);
  } else {
    cb(new Error('Invalid file type!'), false);
  }
};
```

Always validate files to keep your server safe!

UPLOAD TYPES

SINGLE FILE

Upload a single file.

avatar (single file)

```
const upload = multer({
  storage: storage,
  fileFilter: fileFilter
});
app.post('/upload/single',
 upload.single('avatar'), (req, res) => {
  res.json({
    message: 'File uploaded successfully',
    file: req.file
  });
});
```

MULTIPLE FILES (ARRAY)

Upload multiple files (array).

photos (max 5 files)

```
app.post('/upload/multiple',
upload.array('photos', 5), (req, res) => {
  res.json({
    message: 'Files uploaded successfully',
    files: req.files
  });
});
```

MULTIPLE FIELDS

Upload files for multiple fields.

avatar (1 file)

gallery (max 5 files)

```
app.post('/upload/fields', upload.fields([
  { name: 'avatar', maxCount: 1 },
  { name: 'gallery', maxCount: 5 }
]), (req, res) => {
  res.json({
    message: 'Files uploaded',
    files: req.files
  });
});
```

OTHER OPTIONS

limits	Set size limits (fileSize, files, fields, etc.)
dest	Specify destination folder (Disk Storage only)
preservePath	Preserve full path of files (use carefully)
storage	Choose storage engine (diskStorage / memoryStorage)

LIMIT FILE SIZE EXAMPLE

```
const upload = multer({
  storage: storage,
  limits: { fileSize: 5 * 1024 * 1024 } // 5MB
});
```

REQUEST & RESPONSE OBJECTS

req.file (single)

- fieldname
- originalname
- encoding
- mimetype
- size
- destination
- filename
- path

req.files (multiple)

- Array of files
- Same properties as req.file

COMMON ERROR HANDLING

```
app.post('/upload', upload.single('file'), (req, res) => {
  if (!req.file) {
    return res.status(400).json({ error: 'No file uploaded' });
  }
  res.json({ message: 'Upload successful', file: req.file });
});
```

Always handle errors and validate input!

SUPPORTED FILE TYPES (EXAMPLE)

Images	jpg, jpeg, png, gif, webp
Documents	pdf, doc, docx, txt
Videos	mp4, avi, mov, mkv
Audio	mp3, wav, ogg
Others	zip, rar, csv, xls, etc.

BEST PRACTICES

- ✓ Validate file type and size
- ✓ Use secure storage (not public folder)
- ✓ Rename files to avoid conflicts
- ✓ Scan files for malware (if needed)
- ✓ Handle errors gracefully
- ✓ Use environment variables for paths

Secure Uploads — Validate & sanitize files

Flexible Storage — Disk or Memory

Multiple Uploads — Single or Multiple Files

File Filtering — Control allowed files

Fast & Efficient — Built on busboy

STATIC FILES IN EXPRESS.JS

Serve static content like CSS, JS, Images and more

HTML | CSS | JavaScript | Images | Public Folder | Assets

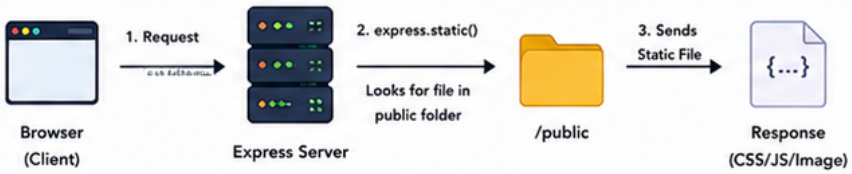
WHAT ARE STATIC FILES?

Static files are files that don't change.
They are sent to tho client exactly as they are.

Examples:

- HTML pages
- CSS stylesheets
- JavaScript files
- Images, Icons, Fonts
- PDF, TXT, etc.

HOW IT WORKS



USING express.static()

```
const express = require('express');
const path = require('path');

const app = express();

// Serve static files from the "public" directory
app.use(express.static(path.join(__dirname, 'public')));

// Server
app.listen(3000, () => {
   console.log('Server running on http://localhost:3000');
});
```

Explanation:

- express.static() is a built-in middleware.
- It serves all files inside the public folder.
- Files are accessible by their path from the browser.

PROJECT STRUCTURE EXAMPLE

```
my-app/
├── app.js
├── package.json
└── public/
    ├── index.html
    ├── css/
    │   └── style.css
    ├── js/
    │   └── script.js
    ├── images/
    │   └── logo.png
    └── assets/
        ├── fonts/
        └── icons/
```

ACCESS IN BROWSER

- http://localhost:3000/index.html — HTML Page
- http://localhost:3000/css/style.css — CSS File
- http://localhost:3000/js/script.js — JavaScript File
- http://localhost:3000/images/logo.png — Image File
- http://localhost:3000/assets/fonts/myfont.woff2 — Font File

STATIC FILE TYPES

HTML | CSS | JavaScript | Images | Fonts | Videos | Audio | Documents

INDEX FILE SUPPORT

When a client requests a folder, Express looks for an "index.html" file inside that folder.

Example:

http://localhost:3000/ → public/index.html (Automatically served)

CUSTOM STATIC PATH (OPTIONAL)

You can serve static files from a custom path.

```
app.use('/assets', express.static(path.join(__dirname, 'public/assets')));
```

Now files inside public/assets will be accessed like:
http://localhost:3000/assets/your-file.png

CACHE CONTROL (OPTIONAL)

Improve performance by setting cache headers.

```
app.use(express.static('public', {
  maxAge: '7d' // cache for 7 days
}));
```

BEST PRACTICES

- Keep static files inside the public folder.
- Use proper folder structure (css, js, images, assets).
- Minify CSS, JS for better performance.
- Use cache control for better speed.
- Never expose sensitive files in public.

express.static() — Built-in middleware
Public Folder — Store all static files
Easy Access — Access via URL path
Fast & Efficient — No processing, direct response
Secure — Only serve what you expose

TEMPLATE ENGINES

Dynamic HTML with Node.js & Express

EJS | Pug | Handlebars | Render | Views | Layouts | Partials | Dynamic HTML

HOW TEMPLATE ENGINES WORK

Request
User requests a page

Express.js
Handles request and calls res.render()

Template Engine
Processes template with data

Views
Loads template files (ejs, pug, hbs, etc.)

Response
Sends rendered HTML to client

WHAT IS A TEMPLATE ENGINE?

Template engines help you generate HTML by combining templates with data. They make your views dynamic, reusable and maintainable.

POPULAR TEMPLATE ENGINES

EJS

Embedded JavaScript templates.
Simple, powerful and fast.

`<% %>`

Pug

Clean, whitespace syntax. No closing tags.

`p, div, span`

Handlebars

Logic-less templates using {{ }} syntax.
Great for simplicity.

`{{ }}`

EXPRESS.JS SETUP

```
const express = require('express');
const app = express();

// Set view engine
app.set('view engine', 'ejs');   // ejs, pug or hbs
app.set('views', __dirname + '/views');

app.get('/', (req, res) => {
   const data = { title: 'My Site', user: 'Lovejeet' };
   res.render('home', data);     // renders views/home.ejs
});

app.listen(3000, () => }
   console.log('Server running on http://localhost:3000');
});
```

RENDER FLOW

1. **Route** /home
2. **res.render()** with data
3. **Template Engine** processes view
4. **Dynamic HTML** sent to client

EXAMPLE: SAME PAGE IN DIFFERENT ENGINES

EJS (home.ejs)

```
<h1><%= title %></h1>
<p>Welcome, <%= user %>!</p>
<% if (user) { %>
   <p>You are logged in.</p>
<% } %>
```

Pug (home.pug)

```
h1= title
p Welcome, #{user}!
if user
   p You are logged in.
```

Handlebars (home.hbs)

```
<h1>{{title}}</h1>
<p>Welcome, {{user}}!</p>
{{#if user}}
   <p>You are logged in.</p>
{{/if}}
```

LAYOUTS & PARTIALS

Layout (views/layout.ejs)

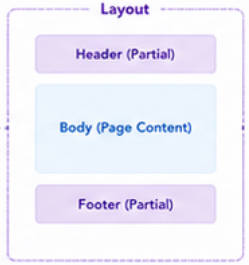
```
<!DOCTYPE html>
<html>
 <head>
   <title><%= title %></title>
   <link rel="stylesheet" href="/css/style.css">
 </head>

 <body>
   <%- include('partials/header') %>
   <main>
     <%- body %>
   </main>
   <%- include('partials/footer') %>
 </body>
</html>
```

Partial (views/partials/header.ejs)

```
<header>
  <nav>
    <a href="/">Home</a> |
    <a href="/about">About</a>
  </nav>
</header>
```

Used in layout:

```
<%- include('partials/header') %>
```



DATA PASSING EXAMPLE

```
app.get('/user/:id', (req, res) => {
  const user = {
    id: req.params.id,
    name: 'Lovejeet Singh',
    role: 'Developer'
  };
  res.render('user', {
    title: 'User Profile',
    user
});});
```

In Template:

```
<h2><%= user.name %></h2>
<p>Role: <%= user.role %></p>
```

WHY USE TEMPLATE ENGINES?

- Keep HTML separate from logic
- Reuse layouts & partials
- Write clean and maintainable code
- Pass dynamic data easily
- Better developer productivity

QUICK COMPARISON

Engine	Syntax	Features	Best For
EJS	<% %>	Full JS power, easy to learn	Flexibility
Pug	Indentation	Clean, terse syntax	Minimal & Readable
Handlebars	{{ }}	Logic-less, simple	Simple, data-driven

Views Folder — Store your template files

Partials — Reusable components

Layouts — Consistent page structure

Dynamic Content — Data-driven HTML

More Productivity — Write less, do more!

VALIDATION
with express-validator

Validate, sanitize and protect your Express.js apps

express-validator | Required | Email | Password | Min Length | Sanitize | Validation | Errors

HOW VALIDATION WORKS

1. Client Request — User submits form data
2. express-validator — Validates and sanitizes input
3. Check Errors — If errors exist, return them
4. Process Data — If valid, process and save data
5. Response — Send success and save data

WHAT IS EXPRESS-VALIDATOR?

express-validator is a set of middlewares for validating and sanitizing user input. It helps you handle validation in a clean, easy and secure way.

INSTALLATION

```
npm install express-validator
```

BASIC SETUP

```
const express = require('express');
const { body, validationResult } = require('express-validator');
const app = express();

app.use(express.json());

app.post('/register', [
  body('name').notEmpty().withMessage('Name is required'),
  body('email').isEmail().withMessage('Enter a valid email'),
  body('password').isLength({ min: 6 })
    .withMessage('Password must be at least 6 characters')
], (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }
  res.json({ message: 'Validation passed! Data is valid.' });
});
```

COMMON VALIDATION RULES

Required
Field must not be empty

```
body('name')
  .notEmpty()
  .withMessage('Name is required')
```

Email
Must be a valid email

```
body('email')
  .isEmail()
  .withMessage('Enter a valid email')
```

Password (Min Length)
Password must be at least 6 characters

```
body('password')
  .isLength({ min: 6 })
  .withMessage('Min 6 characters')
```

Alphanumeric
Only letters and numbers

```
body('username')
  .isAlphanumeric()
  .withMessage('Only letters
  and numbers allowed')
```

Number
Must be a number

```
body('age')
  .isNumeric()
  .withMessage('Age must be a number')
```

Length Range
Must be between min and max

```
body('bio')
  .isLength({ min: 10, max: 100 })
  .withMessage('Bio must be 10-100 chars')
```

Matches
Must match a pattern

```
body('phone')
  .matches(/^\d{10}$/)
  .withMessage('Enter a valid 10 digit
                                phone')
```

Custom Validator
Use custom validation logic

```
body('age').custom((value) => {
  if (value < 18) throw new Error('Must
  be 18 or older');
  return true;
})
```

Sanitize
Clean and sanitize input

```
body('name')
  .trim()
  .escape()
  .withMessage('Invalid characters')
```

GETTING VALIDATION ERRORS

```
app.post('/register', validations, (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({
      success: false,
      errors: errors.array()
    });
  }
  // If no errors, proceed...
});
```

ERRORS RESPONSE EXAMPLE

```
{
  "success": false,
  "errors": [
    {
      "type": "field",
      "value": "",
      "msg": "Name is required",
      "path": "name",
      "location": "body"
    },
    {
      "type": "field",
      "value": "invalid-email",
      "msg": "Enter a valid email",
      "path": "email",
      "location": "body"
    }
  ]
}
```

WHY VALIDATE?

- Ensure data integrity
- Prevent invalid or malicious data
- Provide meaningful error messages
- Improve security
- Better user experience

SANITIZATION EXAMPLES

Trim (remove spaces)

```
body('name').trim()
```

" John " → "John"

Escape (HTML chars)

```
body('bio').escape()
```

"<script>alert(1)</script>"
-> "<script>alert(1)</script>"

Normalize Email

```
body('email').normalizeEmail()
```

" USER@EXAMPLE.COM "
-> "user@example.com"

To Int (convert)

```
body('age').toInt()
```

"25" → 25

Remove Empty Fields

```
check('phone').optional({
  nullable: true
})
```

BEST PRACTICES

- Always validate and sanitize user input
- Return clear and user-friendly error messages
- Keep validation rules in sa separate file
- Use custom validators for complex rules
- Never trust client-side validation only

Validate Early — Validate as soon as possible
Sanitize Always — Clean user input
Clear Errors — Return meaningful messages
Secure Apps — Prevent bad data
Better UX — Happy users
Clean Code — Maintainable & scalable

SECURITY in Express.js

Secure Your API. Protect Your Users.

SECURE API CHECKLIST

- **Helmet** — Set secure HTTP headers
- **CORS** — Control access to your API
- **Rate Limit** — Prevent abuse & brute force
- **XSS Protection** — Prevent cross-site scripting attacks
- **CSRF Protection** — Prevent cross-site request forgery
- **Sanitize Input** — Clean and validate user input
- **HTTPS** — Encrypt data in transit

WHY SECURITY MATTERS?

- Protect user data and privacy
- Prevent common web vulnerabilities
- Build trust with secure APIs
- Stay safe from attacks and abuse

1. HELMET – SECURE HTTP HEADERS

```
const helmet = require('helmet');
app.use(helmet());
```

Helps protect against:

- XSS attacks
- MIME type sniffing
- Clickjacking
- And many more...

2. CORS – CROSS ORIGIN RESOURCE SHARING

```
const cors = require('cors');
app.use(cors({
  origin: 'https://your-frontend.com',
  methods: ['GET', 'POST', 'PUT', 'DELETE'],
  credentials: true
}));
```

Control which domains can access your API.

3. RATE LIMIT – LIMIT REQUESTS

```
const rateLimit = require('express-rate-limit');
const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100, // limit each IP to 100 requests
  message: 'Too many requests, try again later.'
});
app.use(limiter);
```

Limit the number of requests from an IP to prevent abuse and brute force attacks.

4. XSS – CROSS SITE SCRIPTING

```
Sanitize user input and escape output

const xss = require('xss-clean');
app.use(xss());
```

Prevent attackers from injecting malicious scripts into your pages.

5. CSRF – CROSS SITE REQUEST FORGERY

```
const csrf = require('csurf');
app.use(csrf({ cookie: true }));
```

Protects against authorized users being tricked into performing unwanted actions.

6. SANITIZE INPUT

```
const mongoSanitize = require('express-mongo-sanitize');
app.use(mongoSanitize());
```

Remove malicious characters like `$`, `.`, and other NoSQL injection patterns.

7. HTTPS – ENCRYPT DATA

```
const https = require('https');
const fs = require('fs');

const options = {
  key: fs.readFileSync('key.pem'),
  cert: fs.readFileSync('cert.pem')
};
https.createServer(options, app).listen(443);
```

Always use HTTPS in production to encrypt data between client and server.

COMMON SECURITY THREATS

- **XSS** — Cross-Site Scripting: Inject malicious scripts into pages
- **CSRF** — Cross-Site Request Forgery: Trick users into unwanted actions
- **SQL / NoSQL Injection** — Inject malicious queries to access data
- **DDoS / Brute Force** — Flood API with requests to crash or overload the server
- **Sensitive Data Exposure** — Leaking passwords, keys, or internals

BEST PRACTICES

- Use helmet() for secure headers
- Enable CORS with trusted origins only
- Limit requests with rate-limiting
- Sanitize and validate all user input
- Protect against XSS and CSRF
- Use HTTPS everywhere
- Never expose sensitive data or errors
- Keep dependencies updated

SECURE EXPRESS SETUP EXAMPLE

```
const express = require('express');
const helmet = require('helmet');
const cors = require('cors');
const rateLimit = require('express-rate-limit');
const mongoSanitize = require('express-mongo-sanitize');
const xss = require('xss-clean');

const app = express();

app.use(helmet());
app.use(cors({ origin: 'https://your-frontend.com' }));
app.use(rateLimit({ windowMs: 15*60*1000, max: 100 }));
app.use(mongoSanitize());
app.use(xss());
app.use(express.json());

// Your routes here
app.listen(5000, () => console.log('Secure API running!'));
```

EXTRA SECURITY TIPS

Use environment variables for secrets
Never hardcode secrets in code

Use strong passwords & JWT secrets
Keep secrets long and random

Hide error details in production
Don't expose stack traces

Keep dependencies updated
Run `npm audit` regularly

Backup your data regularly
Be prepared for worst case

Security is not optional.
Build secure APIs. Keep users safe.

Secure Headers via Helmet

Control Access with CORS

Prevent Abuse with Rate Limit

Validate & Sanitize All Inputs

Encrypt Everything Use HTTPS

Secure Today
Safe Tomorrow

LOGGING in EXPRESS.JS

Track requests, errors and performance with powerful logging.

Morgan | Console | Logs | Requests | Errors | Monitoring | Debugging | Performance

HOW LOGGING WORKS

WHY LOGGING IS IMPORTANT?

- ✓ Understand application behavior
- ✓ Track requests and responses
- ✓ Detect and fix errors quickly
- ✓ Monitor performance and response time
- ✓ Debug issues in development & production
- ✓ Maintain security and audit trails

Better Logs Better Apps

1. MORGAN – HTTP REQUEST LOGGER

Morgan is a popular middleware for logging HTTP requests.

```
const morgan = require('morgan');
const express = require('express');
const app = express();

app.use(morgan('dev')); // short colored output
// or
app.use(morgan('combined'));   // common log format
```

Morgan Formats

dev	Colored, concise output (default)
combined	Standard Apache combined log format
tiny	Minimal output
short	Shorter than combined
common	Common log format

2. SAMPLE LOG OUTPUT (morgan 'dev')

```
GET     /              200   12.345 ms  -
GET     /api/users     200    8.201 ms  512
POST    /api/login     201   15.678 ms  128
PUT     /api/profile/1 200   11.432 ms  256
DELETE  /api/user/1    204    9.876 ms  -
```

Method | URL | Status Code | Response Time | Response Size

3. CUSTOM CONSOLE LOGGING

Use console or custom logger for important events.

```
app.get('/api/data', (req, res) => {
  console.log('ℹ️ Data requested:', req.ip);
  console.log('📦 Params:', req.query);
  res.json({ success: true });
});

app.use((err, req, res, next) => {
  console.error('❌ Error:', err.message);
  next(err);
});
```

4. LOG LEVELS

Different levels help classify log importance.

INFO	General operational messages
WARN	Warning messages
ERROR	Error events
DEBUG	Detailed debug information
VERBOSE	Very detailed logs

5. LOG DESTINATIONS

Store logs where you need them.

Console	View in terminal
File	Write logs to files
Database	Store logs in DB
Cloud	Send logs to cloud services
Monitoring	Use monitoring tools

6. LOG TO FILE (with winston)

Winston is a flexible logging library.

```
const winston = require('winston');
const logger = winston.createLogger({
  level: 'info',
  format: winston.format.combine(
    winston.format.timestamp(),
    winston.format.json()
  ),
  transports: [
    new winston.transports.File({ filename: 'logs/app.log' }),
    new winston.transports.Console()
  ]
});
logger.info('Server started successfully');
logger.error('Something went wrong!');
```

logs/app.log (example)

```
{
  "level": "info",
  "message": "Server started successfully",
  "timestamp": "2025-05-20T10:15:30.1232"
}
{
  "level": "error",
  "message": "Something went wrong!",
  "timestamp": "2025-05-20T10:16:05.4562"
}
```

7. ERROR LOGGING MIDDLEWARE

Catch and log all errors in one place.

```
app.use((err, req, res, next) => {
  console.error('🔴 Error occurred:');
  console.error('Message:', err.message);
  console.error('Stack:', err.stack);
  console.error('URL:', req.originalUrl);
  console.error('Method:', req.method);

  res.status(500).json({
    success: false,
    message: 'Internal Server Error'
  });
});
```

⚠ Log errors with details to fix issues faster.

8. PERFORMANCE LOGGING

Measure response time and performance.

```
app.use((req, res, next) => {
  const start = process.hrtime.bigint();

  res.on('finish', () => {
    const diff = Number(process.hrtime.bigint() -
    start) / 1e6;

    console.log(`⏱ ${req.method ${req.originalUrl -
    ${res.statusCode - ${diff.toFixed(2)} ms`);
  });
  next();
});
```

9. MONITORING & LOG AGGREGATION

Use tools to aggregate and visualize logs.

- **ELK Stack** (Elasticsearch, Logstash, Kibana)
- **Winston** Node.js logging
- **PM2** Process monitoring
- **Datadog** Cloud monitoring
- **Grafana** Metrics visualization

10. BEST PRACTICES

- ✓ Log meaningful information, not everything
- ✓ Use log levels appropriately
- ✓ Never log sensitive data (passwords, tokens)
- ✓ Use timestamps for all logs
- ✓ Rotate and clean old log files
- ✓ Monitor logs in production

COMMON LOGGING FLOW

Better Debugging — Find and fix issues quickly

Performance Insights — Optimize response time

Improved Reliability — Detect problems early

Production Ready — Monitor and scale with confidence

ERROR HANDLING IN EXPRESS.JS

Handle errors gracefully. Keep your app stable.

try/catch | next(error) | Not Found | Server Error | Custom Error | Async Handler | Error Middleware | Logging

ERROR FLOW IN EXPRESS

Client Sends Request → Route / Controller Processing → Error Occurs (Any Stage) → next(error) Called → Error Middleware Handles It → Response Sent to Client

Always use next(error) to pass errors to the error handling middleware.

WHY ERROR HANDLING IS IMPORTANT?

- Prevent server from crashing
- Send meaningful error messages
- Improve user experience
- Easier debugging & monitoring
- Log errors for future analysis

Robust App Happy Users

1. TRY / CATCH (SYNCHRONOUS CODE)

```
app.get('/user', (req, res) => {
  try {
    const user = getUser();
    res.json(user);
  } catch (error) {
    next(error); // pass error to middleware
  }
});
```

2. ASYNC ROUTE HANDLER WITH NEXT(ERROR)

```
app.get('/users', async (req, res, next) => {
  try {
    const users = await getUsers();
    res.json(users);
  } catch (error) {
    next(error); // forward to error middleware
  }
});
```

In async functions, always pass error to next(error) so Express can handle it.

3. ASYNC HANDLER WRAPPER (CLEAN WAY)

```
// utils/asyncHandler.js
const asyncHandler = (fn) => (req, res, next) => {
  Promise.resolve(fn(req, res, next)).catch(next);
};

// Usage
app.get('/products', asyncHandler(async (req, res) => {
  const data = await getProducts();
  res.json(data);
}));
```

No try/catch in every route. Cleaner & better!

4. CUSTOM ERROR CLASS

```
class AppError extends Error {
  constructor(message, statusCode) {
    super(message);
    this.statusCode = statusCode;
    this.status = `${statusCode}`.startsWith('4')
      ? 'fail'
      : 'error';
    Error.captureStackTrace(this, this.constructor);
  }
}
```

```
// Usage
return next(new AppError('User not found', 404));
```

5. ERROR MIDDLEWARE

```
app.use((err, req, res, next) => {
  // Log the error
  console.error('❌ Error:', err.message);
  console.error(err.stack);
  // Default values
  const statusCode = err.statusCode || 500;
  const message = err.message || 'Internal Server Error';

  res.status(statusCode).json({
    success: false,
    status: err.status || 'error',
    message,
    stack: process.env.NODE_ENV === 'production'
      ? undefined
      : err.stack
  });
});
```

Error middleware MUST have 4 arguments: (err, req, res, next) and should be last!

6. 404 - NOT FOUND HANDLING

```
// Handle undefined routes
app.use((req, res, next) => {
  const err = new AppError('Cannot find ${req.originalUrl}
  on this server', 404);
  next(err);
});
```

404
Not Found
Resource is not available.

7. 500 - INTERNAL SERVER ERROR

```
// Any unexpected error
return next(new AppError('Something went wrong on server', 500));
```

500
Server Error
Something went wrong.

8. ERROR TYPES

- **Validation Error (400)** — Invalid input from user
- **Unauthorized (401)** — Authentication required
- **Forbidden (403)** — Access denied
- **Not Found (404)** — Resource not found
- **Server Error (500)** — Something broke on server

9. LOGGING ERRORS

```
const winston = require('winston');

const logger = winston.createLogger({
  level: 'error',
  format: winston.format.combine(
    winston.format.timestamp(),
    winston.format.json()
  ),
  transports: [
    new winston.transports.File({ filename: 'logs/error.log' }),
    new winston.transports.Console()
  ]
});
```

Always log errors for monitoring & debugging.

10. BEST PRACTICES

- Always use next(error) to pass errors
- Use custom error class for consistent errors
- Don't expose stack trace in production
- Log all errors (file / monitoring service)
- Handle 404 for undefined routes
- Use async handler to avoid try/catch in every route

SAMPLE COMPLETE SETUP

```
const express = require('express');
const { asyncHandler } = require('./utils/asyncHandler');
const { AppError } = require('./utils/AppError');

const app = express();

// Route
app.get('/test', asyncHandler(async (req, res) => {
  throw new AppError('This is a test error', 400);
}));
// 404
app.use((req, res, next) => {
  next(new AppError(`Cannot find ${req.originalUrl} on this server`, 404));
});
// Error Middleware
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(err.statusCode || 500).json({
    success: false,
    message: err.message || 'Internal Server Error'
  });
});
```

RESPONSE EXAMPLES

404 Not Found

```
{
  "success": false,
  "message": "Cannot find /abc on this server"
}
```

500 Server Error

```
{
  "success": false,
  "message": "Internal Server Error"
}
```

QUICK REFERENCE

Concept	Usage
try / catch	Handle synchronous errors
next(error)	Pass error to middleware
Async Handler	Use asyncHandler(fn)
Custom Error	new AppError(message, statusCode)
Error Middleware	(err, req, res, next) => {}
404 Handler	Catch undefined routes
Logging	Log errors for monitoring

Handle Errors Don't Ignore Them | Use Middleware The Right Way | Log Everything Important | Keep Users Safe Show Friendly Messages | Monitor & Improve Your Application

A little error handling today saves hours of debugging tomorrow! 💪

API TESTING in Express.js

Test your APIs, verify responses and debug with ease.

Postman

Thunder Client

GET Read Data | POST Create Data | PUT Update Data | DELETE Delete Data | JSON Format | Secure API

API TESTING FLOW

1. Create Request
Set method, URL, Headers, Body

2. Send Request
Send the request to the server

3. Get Response
Server processes and returns data

4. Analyze Response
Check status, body, headers, time

5. Verify & Debug
Validate API and fix if needed

COMMON HTTP METHODS

GET	Retrieve data from the server
POST	Create new resource
PUT	Update existing resource
DELETE	Delete a resource
PATCH	Partially update resource

POSTMAN EXAMPLE

GET http://localhost:3000/api/users Send

Params | Authorization | Headers (2) | Body | Pre-request Script | Tests | Settings

KEY	VALUE
Content-Type	application/json
Authorization	Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6Ik...

Body | Cookies | Headers (6) | Test Results — 200 OK 24 ms 512 B

Pretty | Raw | Preview | Visualize | JSON

```
{
    "success": true,
    "message": "Users fetched successfully",
    "data": [
        {
            "id": 1,
            "name": "Lovejeet Singh",
            "email": "lovejeet@example.com"
        }
}
```

THUNDER CLIENT EXAMPLE

POST http://localhost:3000/api/login Send

Query | Headers | Auth | Body | Tests

JSON | Form | Form-encode | GraphQL | Binary

```
{
  "email": "lovejeet@example.com",
  "password": "123456"
}
```

Response | Headers (6) | Cookies | Test Results — 200 OK 18 ms 398 B

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "user": {
    "id": 1,
    "name": "Lovejeet Singh",
    "role": "admin"
  }
}
```

SAMPLE API REQUESTS

GET	/api/users	Get all users
POST	/api/users	Create new user
PUT	/api/users/1	Update user
DELETE	/api/users/1	Delete user
GET	/api/users/1	Get user by ID

SAMPLE JSON DATA

JSON

```
{
  "name": "Lovejeet Singh",
  "email": "lovejeet@example.com",
  "age": 28,
  "role": "Developer",
  "skills": ["Node.js", "Express.js", "MongoDB"]
}
```

RESPONSE STATUS CODES

200	OK – Request successful
201	Created – Resource created
400	Bad Request – Invalid request
401	Unauthorized – Auth required
404	Not Found – Resource not found
500	Internal Server Error – Server error

HEADERS EXAMPLE

Header	Example Value
Content-Type	application/json
Authorization	Bearer <token>
Accept	application/json
User-Agent	PostmanRuntime/7.36.0

TESTING CHECKLIST

- Verify all HTTP methods
- Check status codes
- Validate response data
- Test edge cases & errors
- Check authentication
- Test with invalid data
- Measure response time

BEST PRACTICES

- Use environment variables for URLs
- Keep APIs versioned (/api/v1/)
- Validate input on both client & server
- Use proper status codes
- Log and monitor API requests
- Document your APIs (Swagger / OpenAPI)

Test APIs Faster, Save Time | Find Bugs Early, Fix Quickly | Improve API Quality, Better Performance | Secure APIs, Protect Data | Better Collaboration, Team Efficiency

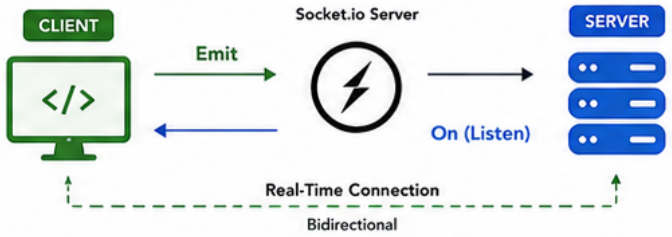
Socket.io

Real-Time Communication

Build real-time applications with WebSockets.

Real-Time | Bidirectional | Low Latency | Scalable | Easy

HOW SOCKET.IO WORKS



KEY FEATURES

- Real-time, bidirectional communication
- Automatic reconnection
- Rooms for group messaging
- Broadcast to multiple clients
- Namespaces for separation
- Acknowledgements
- Scalable and fast

socket.io

CLIENT SIDE (BROWSER)

```
// Include Socket.io client
<script src="/socket.io/socket.io.js"></script>

const socket = io();

// Emit an event
socket.emit('message', 'Hello Server!');

// Listen for an event
socket.on('message', (data) => {
  console.log('Message from server:', data);
});

// Join a room
socket.emit('join', 'room1');
```

SERVER SIDE (NODE.JS)

```
const io = require('socket.io')(server);

// When a client connects
io.on('connection', (socket) => {
  console.log('New client connected:', socket.id);

  // Listen for an event
  socket.on('message', (data) => {
    console.log('Message received:', data);
    // Send back to sender
    socket.emit('message', 'Hello Client!');

    // Broadcast to all other clients
    socket.broadcast.emit('message', '${socket.id}: ${data}');
  });
  // Join room
  socket.on('join', (room) => {
    socket.join(room);
    console.log('${socket.id} joined ${room}');
  });
})
```

EMIT VS ON

EMIT (Send)

Used to send data or trigger an event.

```
socket.emit('event', data);
```

Example:

```
socket.emit('chat', 'Hi!');
```

ON (Listen)

Used to listen for an event.

```
socket.on('event', callback);
```

Example:

```
socket.on('chat', (msg) => {
 console.log(msg);
});
```

BROADCASTING

To all clients (including sender)

```
io.emit('update', data);
```

To all except sender

```
socket.broadcast.emit('update', data);
```

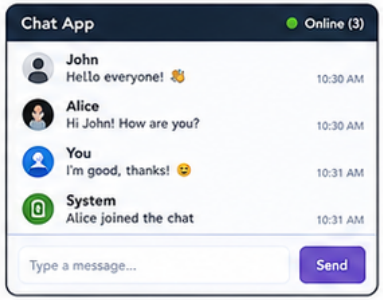
To a specific room

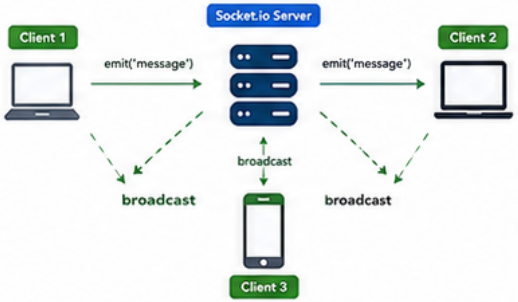
```
io.to('room1').emit('update', data);
```

To all in room except sender

```
socket.to('room1').emit('update', data);
```

REAL-TIME CHAT APP EXAMPLE





COMMON EVENTS

- **connection** — A new client connects
- **disconnect** — A client disconnects
- **message** — Custom message event
- **join** — Join a room
- **leave** — Leave a room
- **typing** — User is typing...
- **stop-typing** — User stopped typing
- **userlist** — Get list of users
- **update** — Send updates
- **error** — Send error message

BEST PRACTICES

- Always handle disconnections
- Validate data on server
- Use rooms for groups
- Keep events simple and meaningful
- Handle errors and timeouts
- Avoid memory leaks (remove listeners)
- Use namespaces for large apps

Real-Time Experience
Instant updates, no refresh needed

Bidirectional
Client & server can talk at the same time

Scalable
Handles thousands of connections easily

Low Latency
Fast and efficient communication

Cross Platform
Works on web, mobile, desktop, IoT

Perfect For
Chat apps, live notifications, gaming, dashboards & more

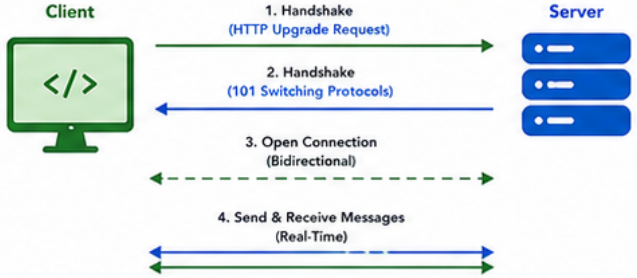
WebSockets

Real-Time, Bidirectional Communication

WebSockets provide a persistent connection between client and server.

- **Real-Time** — Instant updates
- **Bidirectional** — Two-way communication
- **Low Latency** — Very fast performance
- **Persistent** — Single connection, open until closed
- **Scalable** — Efficient for many clients

HOW WEBSOCKETS WORK

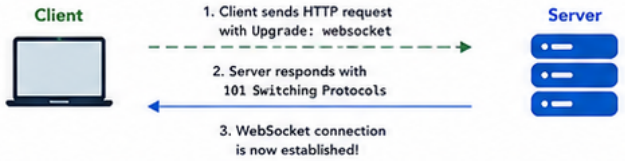


KEY FEATURES

- Persistent connection after handshake
- Full-duplex (bidirectional) communication
- Real-time data transfer
- Event-driven messaging
- Push data from server to client
- Great for live apps (chat, gaming, dashboards)
- Lower overhead than HTTP polling

WebSocket

1. HANDSHAKE PROCESS



Example Handshake Request

```
GET /chat HTTP/1.1
Host: example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNbXBsZSBub2SjZQ==
Sec-WebSocket-Version: 13
```

Example Handshake Response

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

2. BIDIRECTIONAL COMMUNICATION

Message Types

- **Text** (String)
- **Binary** (ArrayBuffer)
- **Ping / Pong** (Keep-alive)
- **Close** (End connection)

3. SIMPLE EXAMPLE (JAVASCRIPT)

```
// Create WebSocket connection
const socket = new WebSocket('wss://example.com/socket');

// Connection opened
socket.onopen = () => {
  console.log('✅ Connected to server');
  socket.send('Hello Server!');
};

// Listen for messages
socket.onmessage = (event) => {
  console.log('📩 Message:', event.data);
};

// Handle errors
socket.onerror = (error) => {
  console.error('❌ Error:', error);
};

// Connection closed
socket.onclose = () => {
  console.log('🔌 Connection closed');
};
```

4. NODE.JS WEBSOCKET SERVER (ws)

```
const WebSocket = require('ws');
const wss = new WebSocket.Server({ port: 8080 });

wss.on('connection', (ws) => {
  console.log('New client connected');

  ws.on('message', (message) => {
    console.log('Received:', message.toString());
    ws.send('Message received!');  // send back
  });

  ws.on('close', () => {
    console.log('Client disconnected');
  });

  // Broadcast message to all clients
  function broadcast(data) {
  wss.clients.forEach(client => {
    if (client.readyState === WebSocket.OPEN) {
      client.send(data);
    }
  });
}
});
console.log('WebSocket server running on ws://localhost:8080');
```

5. COMMON EVENTS

open	Connection is established
message	A message is received
error	An error occurred
close	Connection is closed
ping	Keep-alive signal
pong	Response to ping

6. USE CASES

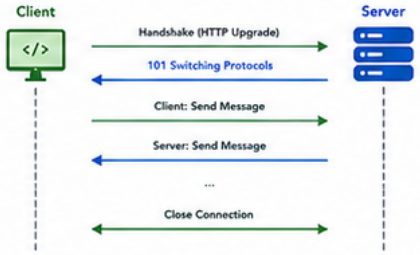
- Live Chat Applications
- Real-Time Notifications
- Online Gaming
- Live Sports / Scores
- Collaborative Tools
- IoT / Sensor Data

7. WEBSOCKET vs HTTP POLLING

Feature	WebSocket	HTTP Polling
Connection	Persistent	New request each time
Communication	Bidirectional	Mostly Client → Server
Latency	Low	High
Overhead	Low	High
Real-Time	Yes	No (Delayed)
Efficiency	High	Low

WebSocket is better for real-time applications!

8. MESSAGE FLOW EXAMPLE



BEST PRACTICES

- Always handle errors and reconnections
- Use ping/pong to keep connection alive
- Close unused connections
- Validate and sanitize incoming messages
- Use secure WebSocket (wss://) in production
- Limit message size
- Handle high concurrency with care

Real-Time Power — Instant data, everywhere

Two-Way Communication — Send & receive in real-time

Lower Latency — Faster than polling

Push Data — Server can push anytime

Reliable & Scalable — Built for modern apps

Perfect For — Live apps & experiences

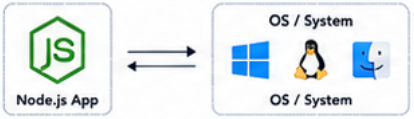
CHILD PROCESS in Node.js

Run system commands, scripts and automate tasks from your Node.js apps.

Run Commands | Multiple Processes | IPC Communication | Automation | Scheduled Tasks

WHAT IS child_process?

The child_process module allows you to spawn subprocesses to run commands, scripts or other programs.



HOW IT WORKS

- Creates a new process
- Runs command / script
- Returns output (stdout / stderr)
- Process exits with code

KEY FEATURES

- Run system commands
- Execute scripts or programs
- Communicate via stdio / IPC
- Easy process management
- Cross-platform support
- Great for automation & tasks

Powerful Automation with child_process

1. exec() – BUFFER OUTPUT

Use case:
Run a command and buffer the entire output.

```
const { exec } = require('child_process');

exec('dir', (err, stdout, stderr) => {
  if (err) {
    console.error('Error: ${err.message}');
    retuun;
  }

  console.log('Output:\n', stdout);
  console.error('Error Output:\n', stderr);
});
```

- Buffers the output in memory
- Simple to use
- Not suitable for large output

2. spawn() – STREAM OUTPUT

Use case:
Run a command and stream the output.

```
const { spawn } = require('child_process');
const ls = spawn('ls', ['-la']);

ls.stdout.on('data', (data) => {
  console.log('stdout: ${data}');
});

ls.stderr.on('data', (data) => {
  console.error('stderr: ${data}');
});

ls.on('close', (code) => {
  console.log('Process exited with code ${code}');
});
```

- Streams output (better for large data)
- More control over the process
- Ideal for long-running commands

3. fork() – NODE.JS SUBPROCESS

Use case:
Spawn a new Node.js process and communicate via IPC.

```
const { fork } = require('child_process');

const child = fork('./worker.js');

child.send({ msg: 'Hello from parent' });

child.on('message', (message) => {
  console.log('Message from child:', message);
});

child.on('exit', (code) => {
  console.log('Child exited with code ${code}');
});
```

- Forks a new Node.js process
- Built-in IPC (process.send / message)
- Best for Node.js scripts

4. execFile() – EXECUTE FILE

Use case:
Run an executable file without using a shell.

```
const { execFile } = require('child_process');

execFile('node', ['-v'], (err, stdout, stderr)
=> {
  if (err) {
    console.error('Error: ${err.message}');
    retoun;
  }

  console.log('Node version:', stdout.trim());
});
```

- More secure (no shell)
- Good for executing files
- Better performance

5. COMPARISON TABLE

Method	Uses Shell	Output Handling	Best For
exec()	Yes	Buffered	Small output, simple commands
spawn()	No	Streamed	Large output, long-running
fork()	No	IPC (JSON)	Node.js processes, communication
execFile()	No	Buffered	Executables, better security

EXAMPLE: RUN SYSTEM COMMANDS

Windows	dir ipconfig
Linux / macOS	ls -la pwd
Node.js	node script.js npm list

COMMUNICATION WITH fork()

parent.js

```
const { fork } = require('child_process');
const child = fork('./child.js');

child.send({ task: 'start' });

child.on('message', (msg) => {
  console.log('Parent received:', msg);
});
```

child.js

```
process.on('message', (msg) => {
  console.log('Child received:', msg);

  process.send({ result: 'Task Completed' });
});
```

COMMON EVENTS

'close'	Process closed
'exit'	Process exited (code)
'error'	Failed to start process
'message'	Message from child (IPC)
'data'	Data from stdout / stderr

BEST PRACTICES

- Use exec() for small commands
- Use spawn() for large output
- Use fork() for Node.js IPC
- Handle errors and exit codes
- Always close processes
- Sanitize inputs (avoid injection)

USE CASES

- Run system commands
- Automation scripts
- Scheduled tasks
- Build tools / CLI apps
- Process large files
- DevOps / CI/CD pipelines

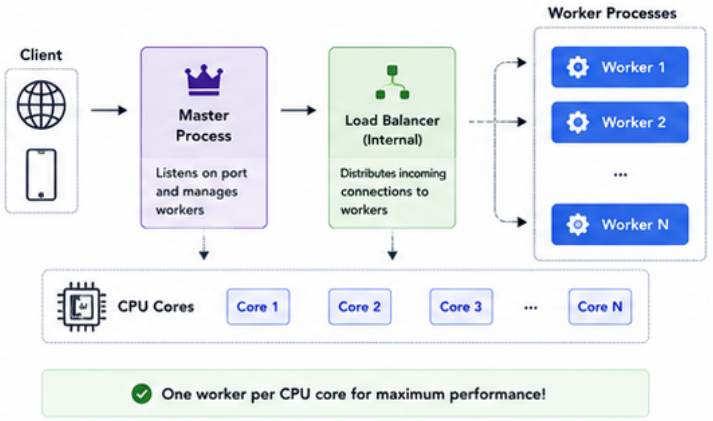
Run Any Command From Your App | Real-Time Output With spawn() | Communicate Easily With fork() | Secure Execution With execFile() | Automate Everything Save Time & Effort | Powerful. Flexible. Fast. That's child_process!

CLUSTER MODULE in Node.js

Run Node.js applications across multiple CPU cores to improve performance, scalability and reliability.

- **Multi Core** — Utilize all CPU cores
- **Load Balancing** — Distribute incoming connections
- **Scalability** — Handle more traffic efficiently
- **High Performance** — Better speed and resource usage

HOW CLUSTER WORKS



One worker per CPU core for maximum performance!

KEY BENEFITS

- **Better Performance** — Utilize all CPU cores for faster response
- **Load Balancing** — Automatically distribute connections
- **Scalability** — Easily handle more users and traffic
- **Fault Tolerance** — If a worker dies, others keep running
- **Efficient Resource Usage** — Optimal use of system resources

More Users.
More Power.
Better Performance.

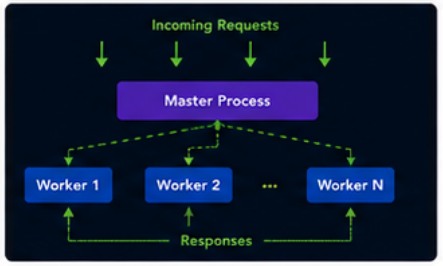
USE CASES

Web Servers · API Servers · Real-time Apps · High Traffic Applications · Microservices

1. BASIC CLUSTER SETUP

```
const cluster = require('cluster');
const os = require('os');
const numCPUs = os.cpus().length;

if (cluster.isMaster) {
  console.log(`Master ${process.pid} is running`);

  // Fork workers.
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }
  // Listen for worker exits
  cluster.on('exit', (worker, code, signal) => {
    console.log(`Worker ${worker.process.pid} died.\
    Starting a new worker...`);
    cluster.fork();
  });
} else {
  // Worker process
  const express = require('express');
  const app = express();
  const port = process.env.PORT || 3000;

  app.get('/', (req, res) => {
    res.send(`Hello from Worker ${process.pid}`)
  });

  app.listen(port, () => {
    console.log(`Worker ${process.pid}
    listening on port ${port}`);
  });
}
```

2. ROUND ROBIN LOAD BALANCING (DEFAULT)



- Requests are distributed evenly (Round Robin)
- Built-in load balancing by Node.js cluster

3. GET CPU CORES

```
const os = require('os');
const numCPUs = os.cpus().length;
console.log(`CPU Cores: ${numCPUs}`);
// Example Output: CPU Cores: 8
```

4. FORK WORKERS DYNAMICALLY

```
if (cluster.isMaster) {
  const numWorkers = os.cpus().length;
  for (let i = 0; i < numWorkers; i++) {
    cluster.fork();
    console.log(`Worker ${i + 1} started`);
  }
}
```

5. WORKER EXIT HANDLING

```
cluster.on('exit', (worker, code, signal) => {
  console.log(`Worker ${worker.process.pid}
  died. Restarting...`);
  cluster.fork();
});
```

- Ensures high availability
- Automatically restarts failed workers

6. CLUSTER EVENTS

Event	Description
fork	Emitted when a new worker is forked
online	Worker is online and ready
listening	Worker is listening for connections
disconnect	Worker disconnected
exit	Worker exited (code, signal)

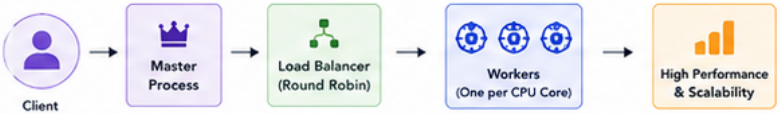
CLUSTER vs SINGLE PROCESS

Feature	Single Process	Cluster (Multi-Process)
CPU Usage	Uses only 1 core	Uses all CPU cores
Performance	Limited	High
Scalability	Low	High
Fault Tolerance	Low (crash = down)	High (others keep running)
Load Balancing	Manual	Built-in (Round Robin)

BEST PRACTICES

- Use cluster for CPU-bound applications
- Keep worker processes stateless
- Handle worker crashes and restarts
- Monitor workers and log errors
- Use PM2 for production process management

CLUSTER FLOW SUMMARY



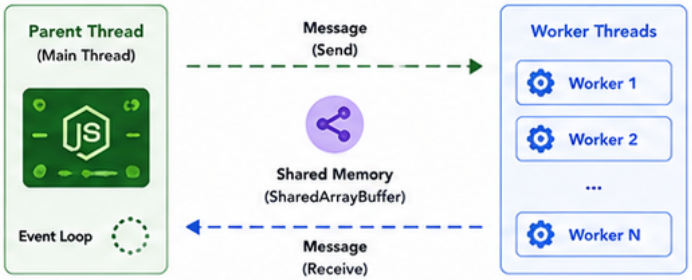
Utilize All CPU Cores — Maximize hardware power
Built-in Load Balancing — Distribute connections evenly
Better Scalability — Handle more users
High Performance — Faster response & throughput
Auto Recovery — Restart workers automatically
Production Ready — Use with PM2 / Docker

WORKER THREADS in Node.js

Run CPU-intensive tasks in parallel without blocking the event loop!

- **Parallel Tasks** — Run multiple tasks at the same time
- **Heavy Computation** — Perfect for CPU intensive work
- **Shared Memory** — Share data using SharedArrayBuffer
- **Better Performance** — Non-blocking, highly efficient

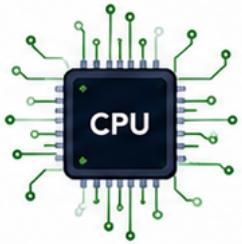
HOW WORKER THREADS WORK



Worker threads run in parallel, share memory, and communicate with the parent thread via messages.

KEY BENEFITS

- **Improved Performance** — Utilize multi-core CPUs effectively
- **Non-Blocking** — Keeps the main event loop responsive
- **Scalability** — Scale CPU-intensive tasks easily
- **Shared Memory** — Exchange data fast using SharedArrayBuffer
- **Better Resource Utilization** — Maximize hardware and boost efficiency



USE ALL CORES.
GET MAX PERFORMANCE!

1. BASIC WORKER THREAD EXAMPLE (PARENT)

```js
// main.js
const { Worker } = require('worker_threads');
const worker = new Worker('./worker.js');
worker.on('message', (message) => {
  console.log('Result from worker:', message);
});
worker.on('error', (err) => {
  console.errer('Worker error:', err);
});
worker.on('exit', (code) => {
  console.log(`Worker stopped with exit code ${code}`);
});
// Send data to worker
worker.postMessage({ value: 10 });
```

2. WORKER THREAD FILE (WORKER)

```js
// worker.js
const { parentPort, workerData } = require('worker_threads');

// Heavy computation
function fibonacci(n) {
  if (n <= 1) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}

parentPort.on('message', (data) => {
  const result = fibonacci(data.value);
  parentPort.postMessage(result); // Send result back
});
```

3. PASS DATA USING workerData

```js
// main.js
const { Worker } = require('worker_threads');

const worker = new Worker('./worker.js', {
  workerData: { num: 20 }
});

worker.on('message', (result) => {
  console.log('Result:', result);
});
```

4. SHARE MEMORY (SharedArrayBuffer)

```js
// main.js
const { Worker } = require('worker_threads');

const buffer = new SharedArrayBuffer(4);
const view = new Int32Array(buffer);
view[0] = 0;

const worker = new Worker('./worker.js', {
  workerData: buffer
});

worker.on('message', () => {
  console.log('Final Value:', view[0]);
});
```

5. COMMON METHODS

Method	Description
new Worker()	Create a new worker thread
postMessage()	Send message to worker
on('message')	Listen for messages from worker
worker.terminate()	Stop the worker thread
worker.unref()	Allow worker to exit if parent exits

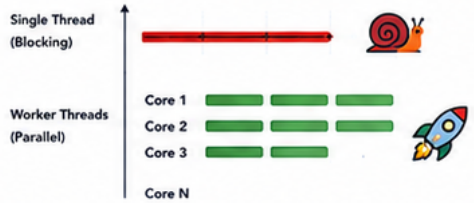
USE CASES

- **Heavy Computations** — Data processing, math, crypto
- **Image / Video Processing** — Resize, convert, filters
- **Machine Learning Tasks** — Model training, predictions
- **Large File Processing** — Parse, transform, analyze
- **Real-time Analytics** — WebSocket data, monitoring

WORKER THREADS vs CLUSTER MODULE

Feature	Worker Threads	Cluster Module
Memory	Share memory	Separate memory
Communication	Fast (Message + Shared Memory)	Slower (IPC)
Overhead	Low	High
Use Case	CPU-intensive tasks	Handle multiple connections
Scaling	Within a process	Multi-process
Startup Time	Faster	Slower

PERFORMANCE IMPACT



BEST PRACTICES

- Use worker threads for CPU-bound tasks
- Limit the number of workers (<= CPU cores)
- Use SharedArrayBuffer for large data
- Handle errors and exit events
- Terminate workers when not needed
- Monitor memory and performance

MAXIMIZE PERFORMANCE
WITH WORKER THREADS!

- **Utilize Multi-Core** — Do more in less time
- **Non-Blocking** — Keep your app responsive
- **Parallel Execution** — Run tasks simultaneously
- **Efficient Communication** — Fast messaging & shared memory
- **Boost Performance** — Better speed, better apps
- **Reliable & Scalable** — Built for production

ADVANCED STREAMS in Node.js

Handle large data efficiently with Streams.
Fast, memory-efficient and scalable!

- **Large Files** — Process big files without loading into memory
- **High Performance** — Lower memory usage and higher throughput
- **Backpressure** — Control data flow automatically
- **Composable** — Pipe multiple streams together

TYPES OF STREAMS

Readable Stream	Writable Stream	Duplex Stream	Transform Stream
Read data from a source.	Write data to a destination.	Read and write data.	Modify or transform data while writing.
Source → App	App → Destination	Two-way communication	Modify data in-flight
fs.createReadStream()	fs.createWriteStream()	net.Socket	zlib, crypto, custom

STREAM PIPELINE

stream.pipeline()

- Automatically handles errors
- Closes all streams on failure
- Best practice for stream flow

KEY BENEFITS

- **Memory Efficient** — Process large data with low memory
- **High Performance** — Better throughput & speed
- **Backpressure Aware** — Prevents overwhelming of data
- **Reliable** — Built-in error handling with pipeline()

1. READABLE STREAM (Read Data)

Read a large file line by line.

```
const fs = require('fs');
const readStream = fs.createReadStream('bigfile.txt');

readStream.on('data', (chunk) => {
   console.log(chunk.toString());
});

readStream.on('end', () => {
   console.log('No more data.\n');
});
```

Use for reading large files, APIs, DB, etc.

2. WRITABLE STREAM (Write Data)

Write data to a file.

```
const fs = require('fs');
const writeStream = fs.createWriteStream('output.txt');

writeStream.write('Hello, Node.js Streams!\n');
writeStream.write('This is line 2.\n');
writeStream.end('Done!\n');

writeStream.on('finish', () => {
   console.log('Write completed.\n');
});
```

Use for writing logs, reports, exports, etc.

3. PIPE (Connect Streams)

Pipe readable stream to writable stream.

```
const fs = require('fs');
const readStream = fs.createReadStream('bigfile.txt');
const writeStream = fs.createWriteStream('copy.txt');

readStream.pipe(writeStream);

writeStream.on('finish', () => {
   console.log('File copied successfully!');
});
```

Easy way to connect readable → writable

4. TRANSFORM STREAM (Modify Data)

Convert text to uppercase while writing.

```
const { Transform } = require('stream');

const upperCase = new Transform({
  transform(chunk, enc, cb) {
     cb(null, chunk.toString().toUpperCase());
  }
});
```

Use for compression, encryption, parsing, etc.

5. BACKPRESSURE (Control Flow)

- If destination is slow, stream pauses source.
- Resumes when destination is ready.

Prevents memory overflow & crashes.

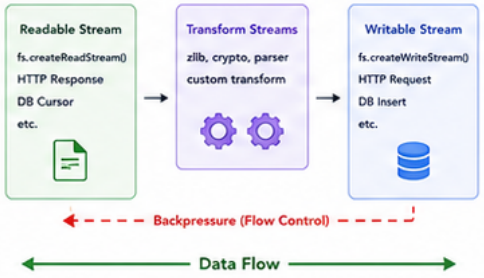
6. PIPELINE (Best Practice)

Use pipeline for better error handling.

```
const fs = require('fs');
const { pipeline } = require('stream');
const zlib = require('zlib');

pipeline(
  fs.createReadStream('bigfile.txt'),
  zlib.createGzip(),
  fs.createWriteStream('bigfile.txt.gz'),
  (err) => {
    if (err) console.error('Pipeline failed:', err);
    else console.log('Pipeline completed!');
  }
);
```

Auto handles errors & closes all streams.

STREAM FLOW DIAGRAM



STREAM EXAMPLE: COMPRESS LARGE FILE

```
const fs = require('fs');
const zlib = require('zlib');
const { pipeline } = require('stream');

pipeline(
  fs.createReadStream('large.log'),
  zlib.createGzip(),
  fs.createWriteStream('large.log.gz'),
  (err) => {
    if (err) console.error('Error:', err);
    else console.log('Compressed successfully!');
  }
);
```

Efficiently compress large files with streams.

COMMON EVENTS

'data'	Emitted when data is available
'end'	No more data to read
'finish'	All data has been written
'error'	Error occurred
'close'	Stream is closed

BEST PRACTICES

- Always prefer pipeline() over pipe()
- Handle errors in streams
- Use proper highWaterMark
- Respect backpressure
- Close streams when done
- Monitor memory usage

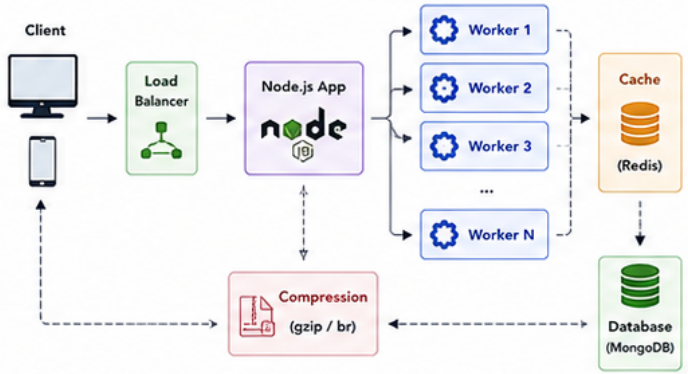
- **Streams = Fast** — Process big data without loading all in RAM
- **Backpressure = Safe** — Automatic flow control prevents overload
- **Pipeline = Reliable** — Built-in error handling & clean up
- **Performance = High** — Better speed, lower memory, high scalability
- **Use Cases** — Logs, backups, uploads, media, ETL, real-time data
- **Master Streams** — Build scalable & high performance apps!

PERFORMANCE OPTIMIZATION in Node.js

Write faster, scalable & efficient Node.js applications

Caching	Compression	Async Code	Cluster	Worker Threads	Memory Usage
Store & reuse frequently used data	Compress responses to reduce size & bandwidth	Use async/await & non-blocking operations	Utilize all CPU cores with cluster module	Run CPU-intensive tasks in parallel threads	Monitor & optimize memory for better performance

HIGH PERFORMANCE ARCHITECTURE



KEY OPTIMIZATION AREAS

- Use caching to reduce database load
- Compress responses (gzip / brotli)
- Write asynchronous & non-blocking code
- Use cluster for multi-core utilization
- Use worker threads for heavy computation
- Optimize memory usage & avoid leaks
- Profile & monitor application performance
- Use efficient algorithms & data structures
- Reduce I/O operations & database queries
- Enable keep-alive & connection pooling

PERFORMANCE METRICS

- **Response Time** — Lower is Better
- **Throughput** — Higher is Better
- **CPU Usage** — Optimize CPU Load
- **Memory Usage** — Keep it Low
- **Event Loop Lag** — Keep it Minimal

1. CACHING (Redis Example)

```
const redis = require('redis');
const client = redis.createClient();

async function getData(key) {
  const cache = await client.get(key);
  if (cache) return JSON.parse(cache);
  const data = await fetchFromDB();
  await client.setex(key, 300, JSON.stringify(data));
  return data;
}
```

Cache frequent data to reduce DB load and improve response time.

2. COMPRESSION (gzip)

```
const express = require('express');
const compression = require('compression');

const app = express();

app.use(compression()); // gzip / br

app.get('/', (req, res) => {
  res.send('Hello, Optimized World!');
});
```

Compress responses to reduce size and increase speed.

3. ASYNC CODE (Non-blocking)

```
const fs = require('fs').promises;

app.get('/', async (req, res) => {
  try {
    const data = await fs.readFile('data.txt','utf8');
    res.send(data);
  } catch (err) {
    res.status(500).send('Error');
  }
});
```

Use async/await to keep event loop free & improve scalability.

4. CLUSTER (Multi-core)

```
const cluster = require('cluster');
const os = require('os');

if (cluster.isMaster) {
  const numCPUs = os.cpus().length;
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }
} else {
  require('./app'); // worker code
}
```

Utilize all CPU cores for more requests & better performance.

5. WORKER THREADS (Heavy Tasks)

```
const { Worker } = require('worker_threads');

function runTask(data) {
  return new Promise((resolve, reject) => {
    const worker = new Worker('./worker.js', {
      workerData: data
    });
    worker.on('message', resolve);
    worker.on('error', reject);
  });
}
```

Run CPU-intensive tasks in parallel without blocking the main thread.

6. MEMORY OPTIMIZATION

```
// Avoid global variables & memory leaks

// Use streams for large files
const stream = fs.createReadStream('big.log');

stream.on('data', chunk => {
  // process chunk
});
stream.on('end', () => console.log('Done'));

};
```

Use streams, clean references & monitor memory usage regularly.

7. PROFILING (Identify Bottlenecks)

```
# CPU Profiling
node --prof app.js
node --prof-process isolate-*.log > profile.txt

# Heap Snapshot
node --inspect app.js
chrome://inspect
```

Profile your app to find slow functions and memory leaks.

8. MONITOR PERFORMANCE

PM2	clinic.js	Prometheus	Grafana
Process Manager	Performance Monitoring	Metrics Collection	Visualization Dashboard

Monitor metrics, logs & alerts to keep your app healthy.

9. OPTIMIZATION CHECKLIST

- Enable caching
- Compress responses
- Use async code
- Use cluster / worker threads
- Optimize memory usage
- Profile & monitor
- Reduce DB queries
- Use efficient algorithms

10. PERFORMANCE IMPACT EXAMPLE

Optimization	Response Time	Throughput	CPU Usage	Memory Usage
Without Optimization	1200ms	500 req/s	90%	350MB
With Optimization	200ms	2500 req/s	40%	120MB
Improvement	83% Faster	5x Higher	55% Lower	65% Lower

BEST PRACTICES

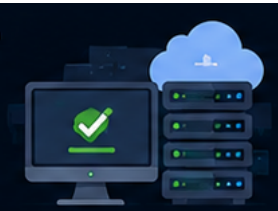
- Write Clean & Efficient Code
- Use Connection Pooling for DB
- Avoid Blocking Operations
- Set Timeouts & Limits
- Handle Errors Gracefully
- Keep Dependencies Updated

> *"Optimize Today, Scale Tomorrow!"*

- Faster Response
- Better Scalability
- Lower Costs
- Happy Users

DEPLOYMENT in Node.js

Take your Node.js app from local to production – fast, secure & scalable!



Production	Node Server	VPS / Cloud	Deploy	Monitor	Scale
Live environment for real users	Run your Node.js application	Virtual private server full control	Upload, configure & go live	Keep your app healthy	Handle more users with ease

DEPLOYMENT FLOW

Code	Build	Upload	Run	Live
Write your Node.js app	Install deps, build (if needed)	Push code to server/platform	Start your app on server	Your app is now live for users

What You Need

- Node.js Application
- Server / Platform
- Environment Variables
- Domain (Optional)

Security Essentials

- Use HTTPS
- Secure Environment Variables
- Firewall & DDoS Protection
- Regular Updates & Backups

POPULAR PLATFORMS

Render
Easy full-stack deployment with free SSL & autoscale.

Railway
Developer friendly, fast deployments.

Vercel
Best for Node.js APIs & frontends (Serverless).

DigitalOcean / VPS
Full control, scalable VPS for any Node.js app.

You Deploy, We Scale!

- High Performance
- 99.9% Uptime
- Global CDN
- Auto Scaling
- Easy Rollbacks

ENVIRONMENT & CONFIGURATION

```
.env

PORT=5000
NODE_ENV=production
MONGO_URI=mongodb://user:pass@host:27017/db
JWT_SECRET=your_jwt_sercet
CORS_ORIGIN=https://yourdomain.com
```

- Use .env for environment variables
- Never commit .env to Git
- Use different configs for development & production

DEPLOY ON VPS (Example: Ubuntu)

```
# 1. Connect to your VPS
ssh user@your_server_ip

# 2. Install Node.js & npm
curl -fsSL https://deb.nodesource.com/setup_20.x | sudo -E bash -
sudo apt-get install -y nodejs

# 3. Clone your app
git clone https://github.com/your-username/your-app.git
cd your-app

# 4. Install dependencies
npm install --production

# 5. Start with PM2
pm2 start app.js --name my-app
pm2 startup && pm2 save
```

- Use PM2 to keep your app running forever.

DEPLOY ON RENDER

- Connect GitHub Repo
- Choose Web Service
- Set Build Command: `npm install`
- Set Start Command: `npm start`
- Add Environment Variables
- Deploy 🚀

Start Command Example

```
npm start
# or
node app.js
```

DEPLOY ON RAILWAY

- Connect GitHub Repo
- Add Environment Variables
- Select Root Directory (if needed)
- Deploy Automatically
- Get Live URL 🚀

Environment Variables

```
PORT=5000
NODE_ENV=production
MONGO_URI=your_mongo_uri
```

DEPLOY ON VERCEL (FOR NODE API)

- Import GitHub Repository
- Select Framework Preset: Other
- Build Command: (leave empty)
- Output Directory: (leave empty)
- Add Environment Variables
- Deploy 🚀

Vercel Config (vercel.json)

```
{
  "version": 2,
  "builds": [{ "src": "app.js", "use": "@vercel/node" }],
  "routes": [{ "src": "/(.*)", "dest": "app.js" }]
}
```

BEST PRACTICES

- Use HTTPS everywhere
- Keep dependencies updated
- Use PM2 or systemd to keep app alive
- Enable CORS only for trusted domains
- Use compression & caching
- Monitor logs & errors
- Backup your database regularly

PM2 – PROCESS MANAGER

```
# Start app
pm2 start app.js --name my-app

# List processes
pm2 list

# Restart app
pm2 restart my-app

# Logs
pm2 logs my-app

# Save & Startup
pm2 save
pm2 startup
```

PM2
Keep your app alive forever!

DEPLOYMENT CHECKLIST

- Code tested locally
- Environment variables set
- Dependencies installed
- Database connected
- Build / Start command working
- Logs are clean
- Domain & SSL configured
- Backup strategy in place

MONITORING & LOGGING

PM2 Monitoring	Winston / Morgan	Sentry
Uptime Robot	Grafana	Health Check API

COMMON ENVIRONMENT VARIABLES

Variable	Description
PORT	Port your app will run on
NODE_ENV	development | production
MONGO_URI	Database connection string
JWT_SECRET	Secret key for JWT
CORS_ORIGIN	Allowed origin for CORS

Deploy Once, Run Anywhere!

Secure Today, Safe Tomorrow!

Monitor, Optimize, Scale Infinite!

Your Code. Live for the World!

Write. Deploy. Inspire.

PM2
PROCESS MANAGER for Node.js

Production process manager for Node.js applications with built-in load balancing and monitoring.

KEY FEATURES

- **Start** — Start your application
- **Restart** — Restart processes
- **Stop** — Stop running processes
- **Logs** — View and manage logs
- **Monitoring** — Real-time monitoring
- **Auto Restart** — Keep apps alive forever
- **Cluster Mode** — Utilize all CPU cores
- **Manage** — Easy process management

PM2 MONITORING DASHBOARD

Overview	
App Name	my-app
Status	online
Uptime	2d 14h 35m
Instances	4
Memory	120.5 MB
CPU	3.2%

CPU Usage 3.2% · Memory Usage 120.5 MB · Event Loop Lag 1.2 ms · Heap Usage 48.7 MB

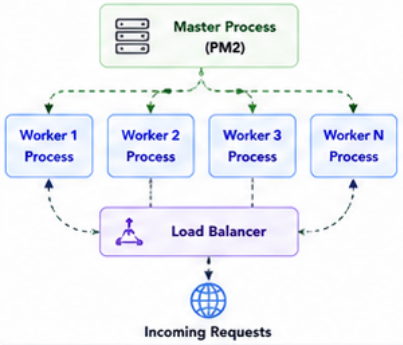
Processes

id	name	mode	status	cpu	memory	uptime
0	my-app	cluster	online	1.3%	52.1 MB	2d 14h
1	my-app	cluster	online	0.8%	47.8 MB	2d 14h
2	my-app	cluster	online	0.6%	49.3 MB	2d 14h
3	my-app	cluster	online	0.5%	48.2 MB	2d 14h

PM2 COMMANDS

Start	`pm2 start app.js`	`# Start application`
Restart	`pm2 restart all`	`# Restart all processes`
Stop	`pm2 stop all`	`# Stop all processes`
Logs	`pm2 logs`	`# View all logs`
Monitor	`pm2 monit`	`# Real-time monitoring`
Cluster Mode	`pm2 start app.js -i max`	`# Start in cluster mode`
Save	`pm2 save`	`# Save process list`
Startup	`pm2 startup`	`# Setup auto start on boot`

CLUSTER MODE (MULTI-CORE)



- **Utilize all CPU cores** — Better performance and scalability
- **Automatic load balancing** — Distribute requests evenly
- **Fault tolerance** — Restart crashed processes
- **Zero downtime reload** — Update without stopping the app

LOGS EXAMPLE

```
pm2 logs

[TAILING] Tailing last 15 lines for [all] processes (change the value with --lines option)
2|my-app  | 2026-07-17 10:30:25 |  info  | Server started on port 3000
2|my-app  | 2026-07-17 10:30:25 |  info  | Environment: production
2|my-app  | 2026-07-17 10:30:26 |  info  | Database connected
0|my-app  | 2026-07-17 10:30:27 |  info  | New user registered
1|my-app  | 2026-07-17 10:30:28 |  warn  | High memory usage detected
3|my-app  | 2026-07-17 10:30:30 |  error | Failed to connect to external API
0|my-app  | 2026-07-17 10:30:32 |  info  | Request processed successfully
2|my-app  | 2026-07-17 10:30:33 |  info  | User logged in
1|my-app  | 2026-07-17 10:30:34 |  info  | Cache cleared
3|my-app  | 2026-07-17 10:30:35 |  info  | File uploaded successfully
```

WHY USE PM2?

- **High Performance** — Optimize and maximize application performance
- **Reliability** — Auto restart on crash and memory limit
- **Scalability** — Scale easily with cluster mode
- **Easy Management** — Simple commands and monitoring
- **Production Ready** — Built for production environments

BEST PRACTICES

- Use ecosystem.config.js for managing apps
- Enable cluster mode in production
- Set memory limits to prevent memory leaks
- Monitor your applications regularly
- Save process list and setup auto startup
- Keep PM2 and dependencies updated

PM2

ECOSYSTEM.CONFIG.JS EXAMPLE

```
module.exports = {
  apps: [{
    name: 'my-app',
    script: 'app.js',
    instances: 'max',
    exec_mode: 'cluster',
    env: {
      NODE_ENV: 'production',
      PORT: 3000
    },
    max_memory_restart: '500M',
    autorestart: true,
    watch: false
  }]
}
```

- **Auto Restart** — Keeps your app alive forever
- **Memory Limit** — Restart app if memory limit is exceeded
- **Zero Downtime** — Reload without downtime

`pm2 start app.js` → `pm2 monit` → `pm2 logs` → `pm2 save` → `pm2 startup`

WRITE ONCE, RUN ANYWHERE!

PM2 makes your Node.js applications production ready.

DOCKER BASICS

Build, Ship & Run Applications Anywhere

Lightweight | Portable | Consistent | Scalable | Fast | Secure

HOW DOCKER WORKS

Image is a blueprint. Container is a running instance of that image.

KEY CONCEPTS

Image
A read-only template with instructions to create a container.

Container
A lightweight, isolated running instance of an image.

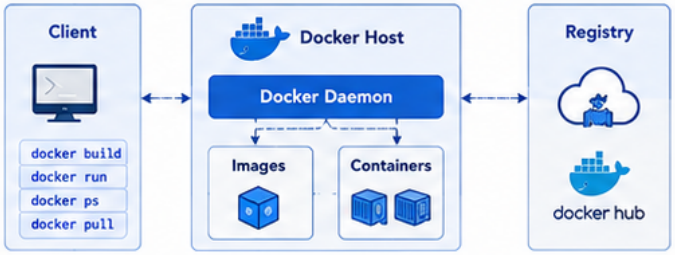
Dockerfile
A text file with commands to build a Docker image.

Volume
Persistent data storage outside the container.

Port Mapping
Map container port to host port to access services.

Registry
Store and share Docker images (Docker Hub).

DOCKER ARCHITECTURE



DOCKERFILE EXAMPLE

```
1  FROM node:20-alpine
2  WORKDIR /app
3  COPY package*.json ./
4  RUN npm install
5  COPY . .
6  EXPOSE 3000
7  CMD ["node", "index.js"]
```

FROM	Base image
WORKDIR	Working directory
COPY	Copy files
RUN	Run commands
EXPOSE	Expose port
CMD	Default command

COMMON DOCKER COMMANDS

```
docker build -t myapp .                    # Build intainer
docker run -d --name mycontainer -p 3000:3000 myapp   # List running containers
docker ps                                  # List alall containers
docker ps -a                               # Stop container
docker stop mycontainer                    # Start container
docker start mycontainer                   # View logs
docker logs mycontainer                    # Remove container
docker rm mycontainer                      # Remove image
docker rmi  myapp                          # List images
docker images                              # Pull image from registry
docker pull  nginx
```

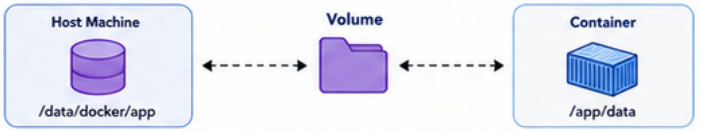
BUILD & RUN WORKFLOW

1. **Write Dockerfile** — Define instructions to build image
2. **Build Image** — Create image from Dockerfile
3. **Run Container** — Run image as container
4. **Access App** — Access your app via mapped port

PORT MAPPING

`-p host_port:container_port` → host_port maps to container_port

VOLUMES (PERSISTENT DATA)



Data in volume remains even if container is stopped or removed.

DOCKER HUB (REGISTRY)

- Push your images
- Share with team
- Pull anywhere
- Public & Private repositories

EXAMPLE: BUILD & RUN NODE.JS APP

```
$ docker build -t node-app .
$ docker run -d --name node-app -p 3000:3000 node-app
$ docker ps
$ docker logs node-app
```

WHY USE DOCKER?

- Consistent environments
- "Works on my machine!"
- Easy deployment
- Resource isolation
- Microservices ready

NOTES

- Containers are isolated but share OS kernel.
- Images are immutable, containers are ephemeral.
- Keep images small for better performance.
- Use .dockerignore to exclude unnecessary files.

BUILD ONCE, RUN ANYWHERE!

Code | Build | Ship | Run | Scale

Docker makes deployment simple, fast & reliable!

TESTING in Node.js

Write tests. Ship with confidence.

Find Bugs Early | Improve Code Quality | Confident Refactoring | Documentation by Examples | Better Maintainability

TYPES OF TESTS

Unit Test
Test small parts of your code in isolation.
Example: Testing a utility function

Integration Test
Test how multiple modules work together.
Example: Testing a service with database

End-to-End Test
Test the complete application flow from start to end.
Example: Testing full API endpoints

EXAMPLE: API TEST WITH SUPERTEST (INTEGRATION TEST)

```
const request = require('supertest');
const app = require('../app');

describe('GET /api/users', () => {
  it('should return users list', async () => {
    const res = await request(app)
      .get('/api/users')
      .expect(200);

    expect(res.body).toBeInstanceOf(Array);
    expect(res.body.length).toBeGreaterThan(0);
  });
});
```

supertest

POPULAR TESTING TOOLS

Jest
- Delightful JavaScript Testing Framework
- Zero config
- Snapshots
- Great DX

Mocha
- Flexible Test Framework
- Simple & powerful
- Works with any assertion library

Chai
- Assertion Library for Node.js
- TDD / BDD friendly
- Should, Expect, Assert styles

Supertest
- HTTP assertions made easy
- Test REST APIs
- Works on top of Superagent

TEST ANATOMY (JEST EXAMPLE)

`describe()`	Group of related tests
`it()` / `test()`	Individual test case
`beforeAll()`	Run once before all tests
`beforeEach()`	Run before each test
`afterEach()`	Run after each test
`afterAll()`	Run once after all tests

```
describe('Math Utils', () => {
  let a, b;

  beforeAll(() => {
    a = 10; b = 5;
  });

  beforeEach(() => {
    // runs before each test
  });

  it('adds numbers', () => {
    expect(a + b).toBe(15);
  });

  it('subtracts numbers', () => {
    expect(a - b).toBe(5);
  });

  afterAll(() => {
    // cleanup
});
```

ASSERTIONS WITH CHAI

```
const { expect } = require('chai');
describe('Chai Assertions', () => {
  it('different assertion styles', () => {
    expect(2 + 2).to.equal(4);          // BDD
    expect('hello').to.be.a('string');// BDD
    expect(true).to.be.true;            // BDD

    expect(5).to.be.greaterThan(2);  // BDD
    expect([1,2,3]).to.include(2);   // BDD

    expect({a:1}).to.have.property('a');// BDD
    const should = require('chai').should(1);
     (2 + 2).should.equal(4);           // TDD
});
```

COMMON MATCHERS
- to.equal()
- to.deep.equal()
- to.be.true / false
- to.be.null / undefined
- to.be.greaterThan()
- to.include()
- to.have.lengthOf()
- to.throw()
- to.match()

RUNNING TESTS (COMMANDS)

`npm test`	Run tests (from package.json "test" script)
`npx jest`	Run Jest tests
`npx jest --watch`	Run tests in watch mode
`npx mocha`	Run Mocha tests
`npx nyc npm test`	Run tests with coverage
`npx jest --coverage`	Run Jest with coverage report

CODE COVERAGE (EXAMPLE)

Statements	Branches	Functions	Lines
92%	85%	90%	93%
184/200	17/20	45/50	186/200

- Identify untested code
- Improve reliability
- Aim for high coverage (80%+)

TESTING BEST PRACTICES

- Keep tests small and focused
- Use descriptive test names
- Follow AAA: Arrange, Act, Assert
- Don't test third-party code
- Mock external dependencies
- Maintain high coverage
- Run tests in CI/CD pipeline

AAA PATTERN

Arrange	Act	Assert
Setup data and context	Execute the function	Verify the outcome

Jest — All-in-one testing framework

Mocha — Flexible & unopinionated test framework

Chai — Powerful assertion library

Supertest — Test HTTP APIs easily

Fast — Quick feedback | Reliable — Catch bugs early | Maintainable — Better code quality | Confident — Ship with confidence

Good tests → Better Code → Happy Users

Node.js PROJECT STRUCTURE

Clean, Scalable & Maintainable Code

- **Organized** — Easy to navigate
- **Modular** — Separation of concerns
- **Scalable** — Easy to grow your app
- **Maintainable** — Easy to maintain & update
- **Best Practice** — Industry standard structure

PROJECT FOLDER

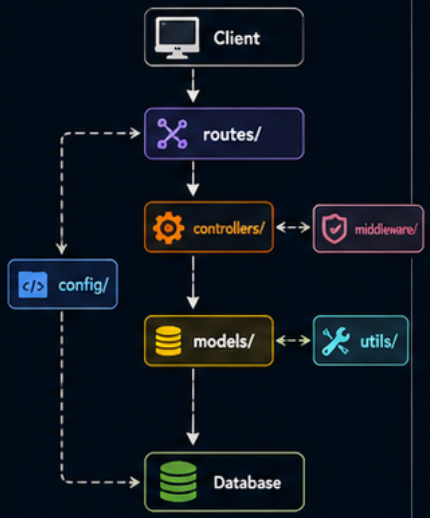
```
my-express-app/
├── node_modules/
├── src/
│   ├── config/
│   ├── routes/
│   ├── controllers/
│   ├── models/
│   ├── middleware/
│   └── utils/
├── .env
├── .gitignore
├── package.json
├── package-lock.json
└── server.js   → Entry Point
```

Keep your project neat and easy to understand for you and your team.

FOLDER & FILES DESCRIPTION

- **config/** — Store configuration files like database, env variables, passport, mail, etc.
- **routes/** — Define all application routes and link them to controllers.
- **controllers/** — Handle request, business logic and send responses.
- **models/** — Define database models and schemas (Sequelize, Mongoose, etc.).
- **middleware/** — Custom middleware functions (auth, error handling, logger, etc.).
- **utils/** — Helper functions and utilities used across the application.
- **.env** — Environment variables and secrets.
- **server.js** — Application entry point. Setup server and start listening.
- **package.json** — Project metadata and dependencies.

STRUCTURE OVERVIEW



SERVER.JS (ENTRY POINT)

```js
require('dotenv').config();
const express = require('express');
const app = express();

const connectDB = require('./src/config/db');
const routes = require('./src/routes');
const errorHandler = require('./src/middleware/
  errorHandler');

app.use(express.json());
app.use('/api', routes);
app.use(errorHandler);

const PORT = process.env.PORT || 5000;
connectDB();
app.listen(PORT, () => {
  console.log('Server running on port ${PORT}');
});
```

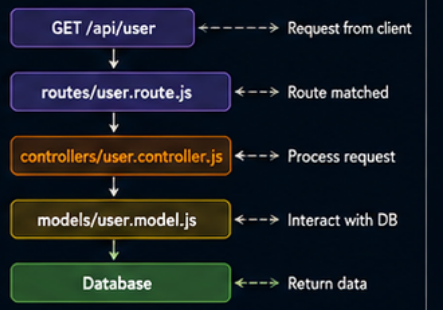
BEST PRACTICES

- Keep controllers thin
- Use meaningful names
- Handle errors properly
- Write reusable code
- Follow consistent structure
- Keep secrets in .env

BENEFITS

- Easy Collaboration
- Faster Development
- Easy Testing
- Better Scalability
- Clean & Professional

EXAMPLE ROUTE FLOW



TIP

A good structure today saves you hours tomorrow!

Organize → Structure → Build → Scale → Succeed

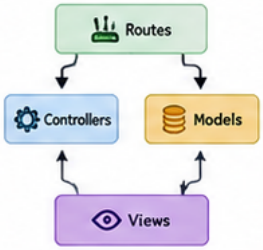
Clean Code. Better Structure. Better Applications.

BEST PRACTICES
for Node.js Applications

Scalable | Secure | Fast | Maintainable | Reliable

Follow Best Practices to build clean, secure, scalable and production-ready Node.js applications.

1 MVC PATTERN



- Separate concerns
- Organized structure
- Easy to test
- Better maintainability

2 CLEAN CODE

```
const getUser = async (id) => {
  try {
    const user = await User.findById(id);
    if (!user) {
      throw new Error('User not found');
    }
    return user;
  } catch (error) {
    throw new Error(error.message);
  }
};
```

- Use meaningful names
- Keep functions small
- Follow consistent formatting
- Add comments where needed

3 REUSABLE MODULES

- Break code into modules
- Reuse across projects
- Easy to maintain and update
- Follow single responsibility

4 ERROR HANDLING

```
try {
  riskyCode();
} catch (err) {
  console.error(err);
  next(err);
}
```

- Handle errors gracefully
- Use try...catch
- Return meaningful messages
- Centralized error handling

5 VALIDATION

- Validate user inputs
- Prevent invalid data
- Use libraries like Joi, express-validator
- Keep data clean and safe

6 SECURITY

- Helmet
- CORS
- Rate Limiting
- Input Sanitization

- Use Helmet for security headers
- Enable CORS carefully
- Prevent NoSQL/ SQL injection
- Limit request rate

7 ENVIRONMENT VARIABLES

```
.env
PORT=5000
NODE_ENV=production
DB_URL=mongodb://localhost:27017/app
JWT_SECRET=your_secret_key
EMAIL_USER=hello@example.com
EMAIL_PASS=your_email_password
```

- Keep sensitive data out of code
- Use .env files
- Different configs for dev, test, production
- Use process.env

8 LOGGING

- Use logging library (Winston, Pino, Bunyan)
- Different log levels
- Logs help in debugging and monitoring
- Store logs safely

EXTRA TIPS

- Backup your database regularly
- Keep dependencies updated
- Write tests for important modules
- Monitor performance and usage
- Follow coding standards
- Write clear documentation

Better Practices
Better Code
Better Applications!

COMMON ERRORS in Node.js

Know the errors, understand the cause and fix with ease.

01 MODULE NOT FOUND

Occurs when Node.js cannot find the required module.

```
Error: Cannot find module 'express'
at Module._resolveFilename (internal/modules/cjs/loader.js:...)
at Module._load (internal/modules/cjs/loader.js:...)
at Module.require (internal/modules/cjs/loader.js:...)
...
```

FIX:

```
npm install <module-name>
```

02 PORT ALREADY IN USE

Occurs when the port you are trying to use is busy.

```
Error: listen EADDRINUSE: address already in use :::3000
at Server.setupListenHandle (node:net:...)
at listenInCluster (node:net:...)
...
```

FIX:

Use another port or stop the process using that port.

```
lsof -i :3000   # mac / linux
netstat -ano | findstr :3000 # win
```

03 CONNECTION FAILED

Occurs when unable to connect to database or external service.

```
Error: connect ECONNREFUSED 127.0.0.1:5432
at TCPConnectWrap.afterConnect [as oncomplete] (node:net:...)
...
```

FIX:

- Check if the service is running.
- Check host, port and credentials.
- Check firewall / network issues.

04 UNHANDLED PROMISE

Occurs when a Promise is rejected but not caught.

```
(node:1234) UnhandledPromiseRejectionWarning:
Error: Something went wrong!
at process.processTicksAndRejections (internal/process/task_queues.js:...)
...
```

FIX:

- Use .catch() to handle rejection.
- Use try...catch with async/await.
- Add global handler for unhandled rejections.

05 SYNTAX ERROR

Occurs when there is a mistake in your code syntax.

```
SyntaxError: Unexpected token '}'
at internalCompileFunction (node:internal/vm:...)
at wrapSafe (node:internal/modules/cjs/loader.js:...)
```

FIX:

Check line number and fix the syntax.
Use a linter or editor to avoid these.

06 REFERENCE ERROR

Occurs when a variable is used before it is defined.

```
ReferenceError: x is not defined
at Object.<anonymous> (app.js:10:5)
at Module._compile (node:internal/modules/cjs/loader.js:...)
...
```

FIX:

Declare the variable before using it.
Check for typos in variable names.

07 TYPE ERROR

Occurs when a value is not of the expected type.

```
TypeError: Cannot read property 'name' of undefined
at Object.<anonymous> (app.js:15:20)
...
```

FIX:

Check if the value exists before using it.
Use optional chaining (?.) to be safe.

BONUS TIPS

- Read the full error message carefully.
- Check the line number mentioned.
- Use console.log() for debugging.
- Restart your server after changes.
- **RTFM:** Read the documentation for better understanding.

ERRORS ARE NORMAL, LEARNING IS IMPORTANT!

Understand the error, fix the cause and write better Node.js applications.

Keep Coding, Keep Learning!

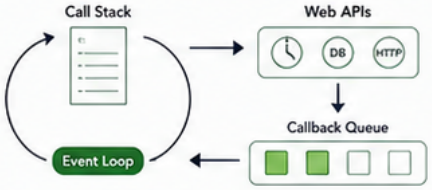
NODE.JS INTERVIEW QUESTIONS

Top Concepts You Must Know

TIPS

- Understand concepts
- Write clean code
- Practice regularly
- Stay consistent

01 EVENT LOOP



Q.
1. What is Event Loop in Node.js?
2. How does Event Loop work?
3. What are phases of Event Loop?
4. What is the role of Callback Queue?

Node.js is single-threaded but non-blocking thanks to the Event Loop.

02 CALLBACK

```
fs.readFile('file.txt', 'utf8',
  function(err, data) {
    if (err) return console.error(err);
    console.log(data);
  }
);
console.log('This will run first');
```

Q.
1. What is a Callback?
2. Why do we use Callbacks in Node.js?
3. What is Callback Hell?
4. How can we avoid Callback Hell?

Callbacks are functions passed as an argument to another function.

03 PROMISE

```
fs.promises.readFile('file.txt', 'utf8')
  .then(data => console.log(data))
  .catch(err => console.error(err));
```

Q.
1. What is a Promise?
2. What are the states of a Promise?
3. How do you handle errors in Promise?
4. What is Promise Chaining?

Promises help to avoid Callback Hell and write cleaner async code.

04 ASYNC AWAIT

```
async function readFileData() {
  try {
    const data = await fs.promises.readFile('file.txt', 'utf8');
    console.log(data);
  } catch (err) {
    console.error(err);
  }
}
```

Q.
1. What is Async Await?
2. How is Async Await different from Promises?
3. Can we use try...catch with Async Await?
4. What happens if we don't use await?

Async Await makes async code look and behave more like synchronous code.

05 STREAMS

Q.
1. What are Streams in Node.js?
2. Types of Streams?
3. What is Pipe in Streams?
4. What is Backpressure?

Streams are used to handle large amount of data efficiently.

06 BUFFER

Q.
1. What is Buffer in Node.js?
2. Why do we need Buffer?
3. How to create a Buffer?
4. How to convert Buffer to String?

Buffer is used to handle binary data directly in memory.

07 EXPRESS

Q.
1. What is Express.js?
2. What is Middleware in Express?
3. How do you handle errors in Express?
4. How to create REST API?

Express.js is a minimal and flexible Node.js web application framework.

08 JWT (JSON WEB TOKEN)

Q.
1. What is JWT?
2. How does JWT work?
3. What are the parts of JWT?
4. How to verify JWT?

JWT is used for authentication and information exchange securely.

PRACTICE. LEARN. BUILD. REPEAT.

Understand Concepts | Write Code | Solve Problems | Mock Interviews | Stay Consistent

Be Confident. Crack the Interview!

Happy Coding!

USEFUL npm PACKAGES

Must-Have Packages for Node.js Development

01 Express

Fast, unopinionated and minimal web framework for Node.js.

```
npm install express
```

```
// Usage
const express = require('express');
const app = express();
app.get('/', (req, res) => res.send('Hello World!'));
app.listen(3000, () => console.log('Server started'));
```

02 Mongoose

Elegant MongoDB object modeling for Node.js.

```
npm install mongoose
```

```
// Usage
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/db');
const userSchema = new mongoose.Schema({
  name: String, email: String
});
const User = mongoose.model('User', userSchema);
```

03 dotenv

Loads environment variables from a **.env** file into process.env.

```
npm install dotenv
```

```
// Usage
require('dotenv').config();
console.log(process.env.PORT);
console.log(process.env.DB_URL);
```

```
.env
PORT=5000
DB_URL=mongodb://
SECRET=xyz123
```

04 bcrypt

Library to help you hash passwords securely.

```
npm install bcrypt
```

```
// Usage
const bcrypt = require('bcrypt');
const hash = await bcrypt.hash(password, 10);
const isMatch = await bcrypt.compare(pass, hash);
```

05 jsonwebtoken

Compact and self-contained way to securely transmit information between parties.

```
npm install jsonwebtoken
```

```
// Usage
const jwt = require('jsonwebtoken');
const token = jwt.sign({ id: userId }, 'secretKey',
  { expiresIn: '1h' });
const decoded = jwt.verify(token, 'secretKey');
```

06 Multer

Middleware for handling multipart/form-data, primarily used for file uploads.

```
npm install multer
```

```
// Usage
const multer = require('multer');
const upload = multer({ dest: 'uploads/' });
app.post('/upload', upload.single('file'),
  (req, res) => res.send('File uploaded'));
```

07 Nodemon

Automatically restarts your Node.js application when file changes.

```
npm install -g nodemon
```

```
// Usage
nodemon server.js
```

Auto Restart on Changes

08 Cors

Middleware to enable Cross-Origin Resource Sharing.

```
npm install cors
```

```
// Usage
const cors = require('cors');
app.use(cors({ origin: 'http://localhost:3000' }));
```

QUICK TIPS

These packages make your Node.js development faster, smarter and more secure!

- Use Express for building APIs quickly.
- Use Mongoose for easy MongoDB interactions.
- Store secrets in .env, never hardcode sensitive data.
- Always hash passwords and use JWT for authentication.
- Handle file uploads safely with Multer.
- Use the right tool for the right job! 🚀

NODE.JS DEVELOPER TOOLS

Top Tools Every Node.js Developer Should Know

01 VS CODE

Powerful code editor with amazing extensions for Node.js development.

- IntelliSense & Auto Completion
- Debugging Support
- Built-in Terminal
- Huge Extension Marketplace

02 NODE.JS

JavaScript runtime built on Chrome's V8 engine to build fast and scalable applications.

```
$ node -v
v20.11.1
$ node app.js
Server running on port 3000
```

- Asynchronous & Event-driven
- Non-blocking I/O
- Perfect for APIs & Real-time Apps
- Large Ecosystem

03 NPM

Node Package Manager for managing packages and project dependencies.

```
$ npm init -y
Wrote to package.json
$ npm install express
added 57 packages, and audited 58 packages
found 0 vulnerabilities
```

- Install & Manage Packages
- Version Control for Packages
- Run Scripts Easily
- Huge Package Registry

04 GIT

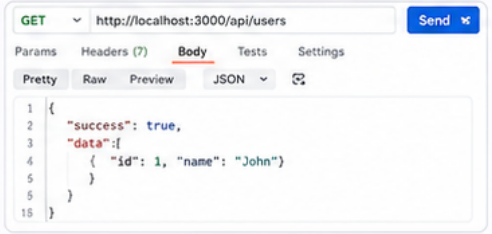
Distributed version control system to track changes and collaborate.

```
$ git status
On branch main
Your branch is up to date with 'origin/main'.
$ git commit -m "Initial commit"
[main (root-commit) a1b2c3d] Initial commit
```

- Track Changes
- Branching & Merging
- Collaboration Made Easy
- History & Rollback

05 POSTMAN

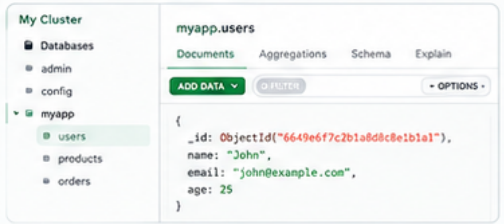
API platform to design, test, and document APIs easily.



- Send HTTP Requests
- Save Collections
- Test APIs
- Environment Variables

06 MONGODB COMPASS

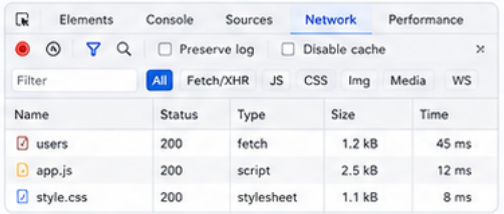
GUI tool to explore, visualize and manipulate MongoDB databases.



- Visualize Data
- Aggregation Builder
- Run Queries
- Performance Insights

07 CHROME DEVTOOLS

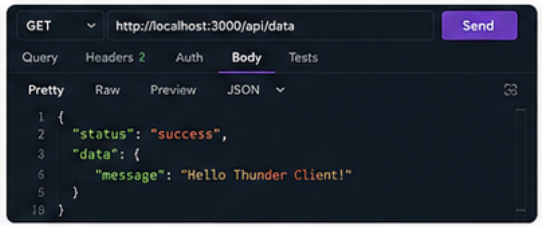
Built-in browser tools for debugging and performance analysis.



- Inspect & Debug
- Performance Analysis
- Network Monitoring
- Console & Breakpoints

08 THUNDER CLIENT

Lightweight REST API client extension for VS Code.



- Send Requests in VS Code
- Environment Support
- Save & Manage APIs
- Fast & Lightweight

PRO TIP
Master these tools to write clean code, debug faster and build better apps! 🚀

Improve Productivity

Debug Efficiently

Write Cleaner Code

Better Collaboration

Build Secure Applications

Great Tools. Better Code. Happy Coding! ♥

NODE.JS DEVELOPMENT WORKFLOW

Step-by-Step Process to Build & Deploy a Node.js Application

01 CREATE PROJECT

Initialize a new Node.js project.

```
$ mkdir my-app
$ cd my-app
$ npm init -y
```

- Create project folder
- Initialize package.json
- Setup basic structure

02 INSTALL PACKAGES

Install required dependencies.

```
$ npm install express
$ npm install mongoose
$ npm install dotenv
```

- Install dependencies
- Save in package.json
- Keep packages updated

03 BUILD API

Create routes, controllers and build your API.

```
app.get('/api/users',
  (req, res) => {
   res.json(users);
});
```

- Create routes
- Add controllers
- Return JSON responses
- Handle errors

04 CONNECT DATABASE

Connect your application to the database.

```
mongoose.connect(process.env.
MONGO_URI)
.then(() => console.log('DB
Connected'))
.catch(err => console.error(err));
```

- Setup MongoDB
- Add connection string
- Handle connection errors
- Use Mongoose models

05 TEST API

Test your API endpoints and verify responses.

```
GET  /api/users  200 OK
{
  "success": true,
  "data": []
}
```

- Use Postman/Thunder Client
- Test all endpoints
- Check status codes
- Validate responses

06 DEBUG

Find and fix bugs to improve performance.

```
console.log('Debug info');
debugger;
// Check variables
// Fix issues
```

- Use Logs
- Use Breakpoints
- Check Errors
- Fix & Optimize

07 DEPLOY

Deploy your application to production.

```
$ git add .
$ git commit -m "Deploy"
$ git push origin main
```

- Push to GitHub
- Choose hosting platform
- Deploy application
- Verify live URL

08 MONITOR

Monitor your app for performance & uptime.

```
Uptime
Errors
Response Time
CPU / Memory
```

- Monitor uptime
- Track errors
- Analyze performance
- Setup alerts

TOOLS YOU WILL USE

VS Code	Node.js	npm	express	MongoDB	Postman	Git	Render / Vercel / Railway	PM2	UptimeRobot
Code Editor	Runtime	Package Manager	Web Framework	Database	API Testing	Version Control	Deployment	Process Manager	Monitoring

BEST PRACTICES

- Write clean and modular code
- Use environment variables
- Follow folder structure
- Handle errors gracefully
- Write tests for APIs
- Keep dependencies updated
- Monitor & optimize regularly

Code. Test. Deploy. Monitor. Repeat.

NODE.JS PROJECT CHECKLIST

Everything You Need for a Production-Ready Node.js Application

01 REST API

- Design clean & resourceful routes
- Use proper HTTP methods
- Return meaningful status codes
- Consistent JSON responses
- Version your API

```
GET     /api/users        Get all users
POST    /api/users        Create user
GET     /api/users/:id    Get user by id
PUT     /api/users/:id    Update user
DELETE  /api/users/:id    Delete user
```

02 AUTHENTICATION

- User registration & login
- Hash passwords (bcrypt)
- Use JWT for authentication
- Protect routes (middleware)
- Handle token expiry

```
const token = jwt.sign(
  { id: user._id },
  process.env.JWT_SECRET,
  { expiresIn: '7d' }
);
```

03 VALIDATION

- Validate request body & params
- Use express-validator / Joi
- Sanitize user inputs
- Return clear error messages
- Prevent invalid data

```
check('email')
  .isEmail().withMessage('Invalid email')
  .normalizeEmail(),
check('password')
  .isLength({ min: 6 });
```

04 DATABASE

- Use Mongoose / ORM
- Define schemas & models
- Validate schema data
- Handle connections properly
- Backup your data

```
const userSchema = new Schema({
  name: String,
  email: { type: String, unique: true },
  password: String
});
```

05 SECURITY

- Use helmet for HTTP headers
- Enable CORS (configure properly)
- Prevent NoSQL injection
- Rate limiting
- Keep dependencies updated

```
app.use(helmet());
app.use(cors({ origin: 'https://yourapp.com' }));
app.use(rateLimit({ windowMs: 15*60*1000, max: 100 }));
```

06 TESTING

- Write unit tests
- Write integration tests
- Test API endpoints
- Use Jest / Mocha
- Aim for good coverage

```
describe('GET /api/users', () => {
  it('should return users', async () => {
    const res = await request(app).get('/api/users');
    expect(res.statusCode).toBe(200);
  });
});
```

07 DEPLOYMENT

- Use environment variables
- Build for production
- Use PM2 / Docker
- Deploy to reliable platform
- Setup CI/CD (optional)

```
npm run build
pm2 start server.js --name my-api
# or using Docker
docker build -t my-api .
docker run -p 3000:3000 my-api
```

08 DOCUMENTATION

- Write clear API docs
- Use Swagger / OpenAPI
- Document request & response
- Include examples
- Keep docs updated

```
{
  "openapi": "3.0.0",
  "info": { "title": "My API", "version": "1.0.0" }
}
```

PRO TIPS

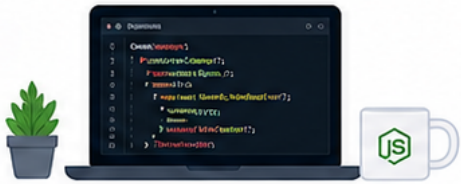
Follow best practices and make your project scalable, secure and maintainable.

- Write clean and modular code
- Handle errors gracefully
- Use logging (Winston / Morgan)
- Monitor performance and optimize
- Keep your dependencies up to date

Check everything off and ship with confidence! 🚀

NODE.JS QUICK REFERENCE

Essential Concepts & Commands at a Glance



01 NODE.JS

- JavaScript runtime
- Event-driven, non-blocking I/O
- Built on Chrome V8 engine
- Perfect for backend APIs

```
// Hello Node.js                    JS
console.log("Hello Node.js!");
console.log(process.version);

node app.js          # Run a file
```

02 MODULES

- Reusability
- Organized code
- Built-in & Custom modules
- CommonJS system

```
// Export                           JS
module.exports = { sayHello };

// Import
const { sayHello } = require('./utils');
```

03 NPM

- Node Package Manager
- Install & manage packages
- Global or local packages
- package.json manages deps

```
# Initialize                       bash
npm init -y
# Install package
npm install express
# Install dev package
npm install -D nodemon
```

04 EXPRESS

- Fast & minimalist web framework
- Routing & middleware
- Easy to build APIs
- Huge ecosystem

```
const express = require('express');                    JS
const app = express();
app.get('/', (req, res) => {
  res.send('Hello Express!');
});
app.listen(3000, () => console.log('Server running'));
```

05 MONGODB

- NoSQL Document Database
- Flexible & scalable
- Works great with Node.js
- Use Mongoose ODM

```
const mongoose = require('mongoose');                  JS
mongoose.connect('mongodb://127.0.0.1:27017/mydb')
  .then(() => console.log('DB Connected'))
  .catch(err => console.error(err));

const User = mongoose.model('User', userSchema);
```

06 JWT

- JSON Web Token
- Secure authentication
- Stateless & scalable
- Used for APIs

```
const jwt = require('jsomoktoken');                    JS
// Sign
const token = jwt.sign({ id: user._id },
 process.env.JWT_SECRET, { expiresIn: '1d' });
// Verify
const data = jwt.verify(token, process.env.JWT_SECRET);
```

07 REST API

- Representational State Transfer
- Use HTTP methods
- Stateless
- JSON data format

Method	Purpose	Example
GET	Get data	/api/users
POST	Create data	/api/users
PUT	Update data	/api/users/1
DELETE	Delete data	/api/users/1

08 DEPLOYMENT

- Deploy to production
- Use environment variables
- PM2 for process management
- CI/CD for automation

```
# Using PM2                        bash
npm install -g pm2
pm2 start app.js --name my-app
pm2 save
pm2 startup
# Deploy to platforms
Render | Railway | Heroku | DigitalOcean
```

PRO TIPS

Write clean code, handle errors, validate inputs, and keep dependencies updated!

- Use environment variables (.env)
- Always validate and sanitize inputs
- Handle errors gracefully
- Use logging for debugging
- Write tests for reliability
- Monitor your app performance

BUILD. TEST. DEPLOY. REPEAT.